

# CCfits Reference Manual

2.2

Generated by Doxygen 1.6.1

Wed Sep 9 11:59:40 2009

---

## Contents

<b>1</b>	<b>CCfits Documentation</b>	<b>1</b>
<b>2</b>	<b>Installing the Package</b>	<b>4</b>
<b>3</b>	<b>Implementation Notes</b>	<b>6</b>
<b>4</b>	<b>Xspec and CCfits</b>	<b>7</b>
<b>5</b>	<b>Getting Started</b>	<b>8</b>
<b>6</b>	<b>Writing Primary Images and Image Extensions</b>	<b>11</b>
<b>7</b>	<b>Creating and Writing to an Ascii Table Extension</b>	<b>13</b>
<b>8</b>	<b>Creating and Writing to a Binary Table Extension</b>	<b>17</b>
<b>9</b>	<b>Copying an Extension between Files</b>	<b>22</b>
<b>10</b>	<b>Selecting Table Data</b>	<b>23</b>
<b>11</b>	<b>Reading Header information from a HDU</b>	<b>24</b>
<b>12</b>	<b>Reading an Image</b>	<b>24</b>
<b>13</b>	<b>Reading a Table Extension</b>	<b>25</b>
<b>14</b>	<b>Reading with Extended File Name Syntax</b>	<b>26</b>
<b>15</b>	<b>What's Present, What's Missing, and Calling CFITSIO</b>	<b>27</b>
<b>16</b>	<b>Previous Release Notes</b>	<b>29</b>
<b>17</b>	<b>Todo List</b>	<b>31</b>
<b>18</b>	<b>Module Index</b>	<b>31</b>
<b>19</b>	<b>Namespace Index</b>	<b>31</b>

<b>1 CCfits Documentation</b>	<b>1</b>
<b>20 Hierarchical Index</b>	<b>32</b>
<b>21 Class Index</b>	<b>34</b>
<b>22 Module Documentation</b>	<b>36</b>
<b>23 Namespace Documentation</b>	<b>38</b>
<b>24 Class Documentation</b>	<b>38</b>

## 1 CCfits Documentation

CCfits-2.2 requires `cfitsio` version 3.08 or later

### 1.1 Introduction

CCfits is an object oriented interface to the `cfitsio` library. `cfitsio` is a widely used library for manipulating FITS (Flexible Image Transport System) formatted files. This following documentation assumes prior knowledge of the FITS format and some knowledge of the use of the `cfitsio` library, which is in wide use, well developed, and available on many platforms.

Readers unfamiliar with FITS but in need of performing I/O with FITS data sets are directed to the first `cfitsio` manual, available at <http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html>. Information about the FITS file format and the current standard is available from <http://fits.gsfc.nasa.gov>

The CCfits library provides an interface that allows the user to manipulate FITS format data through the high-level building blocks of FITS files and Header-Data Units (HDUs). The implementation is designed to hide the details of performing FITS I/O from the user, who will write calls that manipulate FITS objects by passing filenames and lists of strings that represent HDUs, keywords, image data and data columns. Unlike `cfitsio`, which typically requires several calls to access data (*e.g. open file, move to correct header, determine column containing table data, read data*) CCfits is designed to make reading data atomic. For example, it exploits internally existing optimization techniques for FITS I/O, choosing the optimal reading strategy as available [see the `cfitsio` manual, Chapter 13] when data are read on initialization. Data written by CCfits will also be compliant with the FITS standard by specification of class constructors representing FITS dataset elements.

CCfits necessarily works in a fundamentally different way than `cfitsio`. The general pattern of usage for CCfits is: create a FITS object, which either opens a disk file or creates a new disk file, create references to existing or new HDU objects within it, and

manipulated the data through the references. For files with Write access the library is designed to keep the FITS object on disk in sync with the memory copy. The additional memory copy increases the resources required by a calling program in return for some flexibility in accessing the data.

## 1.2 About this Manual

This document lays out the specification for the CCfits library.

The next sections document the installation procedure and the demonstration program *cookbook* which gives examples of usage with comments.

Following sections give a list of what is implemented in CCfits compared to the cfitsio library. For background information and as an example there is a section describing how CCfits is to be used in XSPEC, for which it was originally designed, which may serve to give the reader some insight into the design decisions made.

## 1.3 Release Notes For Version 2.2 Sep 2009

Enhancements to CCfits:

- Added an auto-generated pkg-config file to the stand-alone distribution.
- Added an option for case-insensitive searching in the ExtHDU and Table Column get-by-name functions.
- The public functions `column()` and `numCols()` have been added to the ExtHDU interface. They were previously available only in the derived Table class.
- New `resetRead` and `getNullValue` functions for Column class.
- Improved the documentation for the `nullValue` versions of the Column read/write member functions.

Bug Fixes:

- Converted non-standard calls to vector and valarray end iterators. These were causing runtime exceptions when built with Microsoft Visual C++ v9.0.
- The Column `addNullValue` function now works for cases where the null value argument is of a type that requires casting to match the type of data stored in the Column.
- Fix to the Column `writeArrays` function for the case where valarrays of varying length were sent to fixed-width columns. It was previously sending along a default null value even when the user did not request one.

- Fix for reading and writing complex data types to scalar columns. The first "firstRow" complex values were not being written or read.
- Renamed private FITS::extension() function to extensionMap(). This is to prevent user from having to explicitly declare a const FITS pointer in order to use the public const FITS::extension\*() function.

Backwards Compatibility Issue:

- To prevent overloading ambiguity resulting from the new flag added to ExtH-  
DU/Table get-by-name Column functions for case-insensitive searches, the pro-  
tected column(string,Column\*) function has been renamed to setColumn. As  
this is a protected function, the change should not affect standard usage of CC-  
fits.

## 1.4 Release Notes For Version 2.1 Nov 2008

- Modified several FITS constructors and FITS::open function to allow proper han-  
dling of CFITSIO **extended filename syntax**.
- Extended filename syntax example added to cookbook.
- Fix made to FITS::read function for case of missing EXTVER keyword when  
searching for HDU with extver > 1.
- Removed inclusion of the internal-only CFITSIO fitsio2.h file from Column.cxx.  
LONGLONG limits definitions are now found in fitsio.h.

For a more complete listing, see the CHANGES file distributed with the software. For  
earlier versions, see [Previous Release Notes](#).

## 1.5 Authors and Acknowledgements

CCfits was written as part of a re-engineering effort for the X-Ray data anal-  
ysis program, XSPEC. It was designed using Rational Rose and originally im-  
plemented on a Solaris platform by Ben Dorman to whom blame should be at-  
tached. Sandhya Bansal worked on part of the implementation and, and Paul  
Kunz ([pfkeeb@slac.stanford.edu](mailto:pfkeeb@slac.stanford.edu)) wrote the configuration scheme and dis-  
pensd helpful advice: both are also thanked profusely for the port to Win-  
dows2000/VC++.net. Thanks to R. Mathar (MPIA) and Patrik Jonsson (Lick Obs.)  
for contributing many helpful suggestions and bug reports, and ports to HP-UX and  
AIX respectively.

CCfits is currently maintained by Craig Gordon and Bryan Irby  
([ccfits@heasarc.gsfc.nasa.gov](mailto:ccfits@heasarc.gsfc.nasa.gov)). Suggestions and bug reports are

welcome, as are offers to fill out parts of the implementation that are missing. We are also interested in knowing which parts of cfitsio that are not currently supported should be the highest priority for future extensions.

## 2 Installing the Package

### 2.1 Platforms

CCfits is generally supported on the same platforms as HEASOFT, and on Windows with VC++ 7.0 or later. See the HEASOFT [supported platforms](#) page.

### 2.2 Building

To build and install CCfits from source code on a UNIX-like (e.g. UNIX, Linux, or Cygwin) platform, take the following steps. For building on a Microsoft Windows platform with Visual Developer Studio, see below.

#### 2.2.1 Instructions for Building CCfits on UNIX-like platforms:

##### 1. Configure

By default, the GCC compiler and linker will be used. If you want to compile and link with a different compiler and linker, you can set some environment variable before running the configure script. For example, to use Sun's C++ compiler, do the following:

```
> setenv CXX CC (csh syntax)
```

or

```
> export CXX=CC (bash syntax)
```

You can set the absolute path to the compiler you want to use if necessary.

CCfits requires that the CFITSIO package, version 3.02 or later, is available on your system. See

<http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html>

for more information. The configure script that you will run takes an option to specify the location of the CFITSIO package.

If the CFITSIO package is installed in a directory consisting of a 'lib' subdirectory containing "libcfitsio.a" or "libcfitsio.so" and an 'include' subdirectory containing "fitsio.h", then you can run the configure script with a single option. For example, if the cfitsio package is installed in this fashion in /usr/local/cfitsio/ then the configure script option will be

```
--with-cfitsio=/usr/local/cfitsio
```

If the CFITSIO package is not installed in the above manner, then you need to run the configure script with two options, one to specify the include directory and the other to specify the library directory. For example, if the cfitsio package was built in /home/user/cfitsio/ then the two options will be

```
--with-cfitsio-include=/home/user/cfitsio --with-cfitsio-libdir=/home/user/cfitsio
```

For users of HEASOFT (instead of stand-alone CFITSIO): Note that modern distributions of HEASOFT only include a "libcfitsio\_X.XX.so" library by default, but the configure script needs to find "libcfitsio.so", so you will need to create a symbolic link in \$HEADAS/lib/ linking libcfitsio.so -> libcfitsio\_X.XX.so in order for CCfits to configure properly. You can then configure CCfits using "--with-cfitsio=\$HEADAS/lib".

You have the option of carrying out the build in a separate directory from the source directory or in the same directory as the source. In either case, you need to run the configure script in the directory where the build will occur. For example, if building in the source directory with the cfitsio directory in /usr/local/cfitsio/ then the configure command should be issued like this:

```
> ./configure --with-cfitsio=/usr/local/cfitsio
```

If you do the build in a separate directory from the source, you may need to issue the configure command something like this:

```
> ../CCfits/configure --with-cfitsio=/usr/local/cfitsio
```

The configure script will create the Makefile with the path to the compiler you choose (or GCC by default), and the path to the CFITSIO package. The configure script has other options, such as the install location. To see these options type

```
> ./configure --help
```

## 2. Build

Building the C++ shared library and Java classes will be done automatically by running make without arguments like this:

```
> gmake
```

## 3. Install

To install, type:

```
> make install
```

The default install location will be /usr/local/lib for the library and /usr/local/include for the header files. You can change this with the --prefix option when you configure, or with something like...

```
> make DESTDIR=/usr/local/CCfits install
```

### 2.2.2 Instructions for Microsoft Windows build:

Compiling CCfits with MS VC++ requires VC++ 7.0 or later. This is the compiler that comes with Visual Studio.NET. Earlier versions of the compiler has too many defects in the area of instantiating templates.

Take the following steps.

1. Compile the C++ code. Open the `vs.net/CCfits/CCfits.sln` file with Visual Studio.NET. The includes paths have been set to find the `cfitsio` build directory at the same level as the `CCfits` directory. If this is not the case, use Visual Studio.NET to edit the include paths and extra library paths to where you have `cfitsio` installed.

Next, just use the build icon or the build menu item.

To build the test program, `cookbook`, use the `vs.net/cookbook.cookbook.sln` file

Author: [Paul\\_Kunz@slac.stanford.edu](mailto:Paul_Kunz@slac.stanford.edu) Revised 1 Nov 2006 by Bryan Irby

## 3 Implementation Notes

This section comments on some of the design decisions for CCfits. We note the role of `cfitsio` in CCfits as the underlying "engine," the use of the C++ standard library. We also explain some of the choices made for standard library containers in the implementation - all of which is hidden from the user [as it should be].

Most importantly, the library wraps rather than replaces the use of `cfitsio` library; it does not perform direct disk I/O. The scheme is designed to retain the well-developed facilities of `cfitsio` (in particular, the extended file syntax), and make them available to C++ programmers in an OO framework. Some efficiency is lost over a 'pure' C++ FITS library, since the internal C implementation of many functions requires processing if blocks or switch statements that could be recoded in C++ using templates. However, we believe that the current version strikes a reasonable compromise between developer time, utility and efficiency.

The implementation of CCfits uses the C++ Standard Library containers and algorithms [also referred to as the Standard Template Library, (STL)] and exception handling. Here is a summary of the rationale behind the implementation decisions made.

- HDUs are contained within a FITS object using a `std::multimap<string, HDU*>` object.
  1. The map object constructs new array members on first reference
  2. Objects stored in the map are sorted on entry and retrieved efficiently using binary search techniques.
  3. The pointer-to-HDU implementation allows for polymorphism: one set of operations will process all HDU objects within the FITS file



4. String objects (`char*`) are represented by the `std::string` class, which has a rich public interface of search and manipulation facilities.
- Scalar column data [one entry per cell] are implemented using `std::vector<T>` objects.
  - Vector column data [multiple and either fixed or variable numbers of entries per cell] are implemented using `std::vector<std::valarray<T>>` objects. The `std::valarray` template is intended for optimized numeric processing. `valarrays` have the following desirable features:
    1. they are dynamic, but designed to be allocated in full on construction rather than dynamic resizing during operation: this is, what is usually needed in FITS files.
    2. They have built-in vectorized transcendental functions (e.g. `std::valarray<T> sin(const std::valarray<T>& )`).
    3. They provide `std::valarray<T> apply(T f(const T&))` operation, to apply a function `f` to each element
    4. They provide slicing operations [see the "Getting Started" section for a simple example].
  - Exceptions are provided in for by a `FitsException` hierarchy, which prints out messages on errors and returns control to wherever the exception is caught. Non-zero status values returned by `cfitsio` are caught by subclass `FitsError`, which prints the string corresponding to an input status flag. `FitsException`'s other subclasses are thrown on array bounds errors and other programming errors. Rare [we hope] errors that indicate programming flaws in the library throw `FitsFatal` errors that suggest that the user report the bug.

## 4 Xspec and CCfits

This section is provided for background. Users of CCfits need not read it except to understand how the library was conceived and therefore what its strengths and weaknesses are likely to be in this initial release.

### 4.1 About XSPEC

XSPEC is a general-purpose, multi-mission X-Ray spectral data analysis program which fits data with theoretical models by convolving those models through the instrumental responses. In XSPEC 11.x and all prior versions that use FITS format data, each individual data file format that is supported can have up to 4 ancillary files. That is, for each data file, there can be a response, correction, background and auxiliary response (efficiency) file. Additionally there are table models that read FITS format

data. All told, therefore, much duplicated procedural code for reading FITS data can be eliminated by use of the greater encapsulation provided by CCfits. XSPEC's primary need is to read FITS floating point Binary Tables. XSPEC also creates simulated data by convolving users' models with detector responses, so it also has a need for writing tabular data. Images are not used in XSPEC. We have provided a support for image operations which has undergone limited testing compared to the reading interface for table extensions.

## 4.2 New Data Formats

New formats to be read in XSPEC that are specialized for a particular satellite mission can be supported almost trivially by adding new classes that read data specified with different FITS format files. A single constructor call specifying the required columns and keys is all that is needed to read FITS files, rather than a set of individual cfitsio calls. The library is designed to encourage the "resource acquisition is initialization" model of resource management; CCfits will perform more efficiently if data are read on construction.

## 4.3 Modularity

Third, in an object oriented design, it is possible to make a program only loosely dependent on current implementation assumptions. In XSPEC, data are read as SF and FITS format (of three different varieties) and the user interface is written in tcl/tk. Both of these assumptions could be changed over the future life of the program. Thus the design of XSPEC, and any similar program, consists of defining an abstract DataSet class which has a subclass that uses FITS data. The virtual functions that support reading and writing can easily be overloaded by alternatives to FITS. Thus, the class library specified here fits in with the need for modularity in design.

# 5 Getting Started

The program `cookbook.cxx`, analogous to the `cookbook.c` program supplied with cfitsio, was generated to test the correct functioning of the parts of the library and to provide a demonstration of its usage.

The code for `cookbook` is reproduced here with commentary as worked example of the usage of the library.

## 5.1 Driver Program

```
// The CCfits headers are expected to be installed in a subdirectory of
// the include path.
```

```
// The <CCfits> header file contains all that is necessary to use both the CCfits
// library and the cfitsio library (for example, it includes fitsio.h) thus makin
// g
// all of cfitsio's macro definitions available.

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

// this includes 12 of the CCfits headers and will support all CCfits operations.

// the installed location of the library headers is $(ROOT)/include/CCfits

// to use the library either add -I$(ROOT)/include/CCfits or #include <CCfits/CCf
// its>
// in the compilation target.

#include <CCfits>

#include <cmath>
    // The library is enclosed in a namespace.

    using namespace CCfits;

int main();
int writeImage();
int writeAscii();
int writeBinary();
int copyHDU();
int selectRows();
int readHeader();
int readImage();
int readTable();
int readExtendedSyntax();

int main()
{

    FITS::setVerboseMode(true);

    try

    {

        if (!writeImage()) std::cerr << " writeImage() \n";
        if (!writeAscii()) std::cerr << " writeAscii() \n";
        if (!writeBinary()) std::cerr << " writeBinary() \n";
        if (!copyHDU()) std::cerr << " copyHDU() \n";
        if (!readHeader()) std::cerr << " readHeader() \n";
        if (!readImage()) std::cerr << " readImage() \n";
        if (!readTable()) std::cerr << " readTable() \n";
```

```

        if (!readExtendedSyntax()) std::cerr << " readExtendedSyntax() \n";
        if (!selectRows()) std::cerr << " selectRows() \n";

    }
    catch (FitsException&)
    // will catch all exceptions thrown by CCfits, including errors
    // found by cfitsio (status != 0)
    {

        std::cerr << " Fits Exception Thrown by test function \n";

    }
    return 0;
}

```

The simple driver program illustrates the setting of verbose mode for the library, which makes all internal exceptions visible to the programmer. This is primarily for debugging purposes; exceptions are in some cases used to transfer control in common circumstances (e.g. testing whether a file should be created or appended to in write operations). Most of the exceptions will not produce a message unless this flag is set.

Nearly all of the exceptions thrown by CCfits are derived from FitsException, which is caught by reference in the above example. This includes all nonzero status codes returned by cfitsio by the following construct (recall that in the [cfitsio library](#) nearly all functions return a non-zero status code on error, and have a final argument status of type int):

```

if ( [cfitsio call](args,...,&status)) throw FitsError(status);

```

FitsError, derived from FitsException, uses a cfitsio library call to convert the status code to a string message.

The few exceptions that are not derived from FitsException indicate fatal conditions implying bugs in the library. These print a message suggesting the user contact [HEASARC](#) to report the problem.

Note also the lack of statements for closing files in any of the following routines, The destructor (dtor) for the FITS object does this when it falls out of scope. A call

`FITS::destroy() throw()`

is provided for closing files explicitly; destroy() is also responsible for cleaning up the FITS object and deallocating its resources.

When the data are being read instead of written, the user is expected to copy the data into other program variables [rather than use references to the data contained in the FITS object].

The routines in this program test the following functionality:

`writeImage()` [Writing Primary Images and Image Extensions](#)

`writeAscii()` [Creating and Writing to an Ascii Table Extension](#)

writeBinary() [Creating and Writing to a Binary Table Extension](#)

copyHDU() [Copying an Extension between Files](#)

selectRows() [Selecting Table Data](#)

readHeader() [Reading Header information from a HDU](#)

readImage() [Reading an Image](#)

readTable() [Reading a Table Extension](#)

readExtendedSyntax() [Reading with Extended File Name Syntax](#)

## 6 Writing Primary Images and Image Extensions

This section of the code demonstrates creation of images. Because every fits file must have a PHDU element, all the FITS constructors (ctors) instantiate a PHDU object. In the case of a new file, the default is to establish an empty HDU with BITPIX = 8 (BYTE\_IMG). ***A current limitation of the code is that the data type of the PHDU cannot be replaced after the FITS file is created.*** Arguments to the FITS ctors allow the specification of the data type and the number of axes and their lengths. An image extension of type float is also written by calls in between the writes to the primary header demonstrating switch between HDUs during writes.

Note that in the example below data of type *float* is written to an image of type *unsigned int*, demonstrating both implicit type conversion and the cfitsio extension to unsigned data.

User keywords can be added to the PHDU after successful construction and these will both be accessible as container contents in the in-memory FITS object as well as being written to disk by cfitsio.

Images are represented by the standard library valarray template class which supports vectorized operations on numeric arrays (e.g. taking the square root of an array) and slicing techniques.

The code below also illustrates use of C++ standard library algorithms, and the facilities provided by the std::valarray class.

```
int writeImage()
{
    // Create a FITS primary array containing a 2-D image
    // declare axis arrays.
    long naxis    = 2;
    long naxes[2] = { 300, 200 };

    // declare auto-pointer to FITS at function scope. Ensures no resources
    // leaked if something fails in dynamic allocation.
    std::auto_ptr<FITS> pFits(0);
```

```

try
{
    // overwrite existing file if the file already exists.

    const std::string fileName("!atestfil.fit");

    // Create a new FITS object, specifying the data type and axes for the primary
    // image. Simultaneously create the corresponding file.

    // this image is unsigned short data, demonstrating the cfitsio extension
    // to the FITS standard.

    pFits.reset( new FITS(fileName , USHORT_IMG , naxis , naxes ) );
}
catch (FITS::CantCreate)
{
    // ... or not, as the case may be.
    return -1;
}

// references for clarity.

long& vectorLength = naxes[0];
long& numberOfRows = naxes[1];
long nelements(1);

// Find the total size of the array.
// this is a little fancier than necessary ( It's only
// calculating naxes[0]*naxes[1]) but it demonstrates use of the
// C++ standard library accumulate algorithm.

nelements = std::accumulate(&naxes[0],&naxes[naxis],1,std::multiplies<long>()
);

// create a new image extension with a 300x300 array containing float data.

std::vector<long> extAx(2,300);
string newName ("NEW-EXTENSION");
ExtHDU* imageExt = pFits->addImage(newName,FLOAT_IMG,extAx);

// create a dummy row with a ramp. Create an array and copy the row to
// row-sized slices. [also demonstrates the use of valarray slices].
// also demonstrate implicit type conversion when writing to the image:
// input array will be of type float.

std::valarray<int> row(vectorLength);
for (long j = 0; j < vectorLength; ++j) row[j] = j;
std::valarray<int> array(nelements);
for (int i = 0; i < numberOfRows; ++i)
{
    array[std::slice(vectorLength*static_cast<int>(i),vectorLength,1)] = row
    + i;
}

```

```

// create some data for the image extension.
long extElements = std::accumulate(extAx.begin(),extAx.end(),1,std::multiplie
    s<long>());
std::valarray<float> ranData(extElements);
const float PIBY (M_PI/150.);
for ( int jj = 0 ; jj < extElements ; ++jj)
{
    float arg = PIBY*jj;
    ranData[jj] = std::cos(arg);
}

long fpixel(1);

// write the image extension data: also demonstrates switching between
// HDUs.
imageExt->write(fpixel,extElements,ranData);

//add two keys to the primary header, one long, one complex.

long exposure(1500);
std::complex<float> omega(std::cos(2*M_PI/3.),std::sin(2*M_PI/3));
pFits->pHDU().addKey("EXPOSURE", exposure,"Total Exposure Time");
pFits->pHDU().addKey("OMEGA",omega," Complex cube root of 1 ");

// The function PHDU& FITS::pHDU() returns a reference to the object represen
    ting
// the primary HDU; PHDU::write( <args> ) is then used to write the data.

pFits->pHDU().write(fpixel,nelements,array);

// PHDU's friend ostream operator. Doesn't print the entire array, just the
// required & user keywords, and is provided largely for testing purposes [se
    e
// readImage() for an example of how to output the image array to a stream].

std::cout << pFits->pHDU() << std::endl;

return 0;
}

```

## 7 Creating and Writing to an Ascii Table Extension

In this section of the program we create a new Table extension of type AsciiTbl, and write three columns with 6 rows. Then we add another copy of the data two rows down (starting from row 3) thus overwriting values and creating new rows. We test the use of null values, and writing a date string. Implicit data conversion, as illustrated for images above, is supported. However, writing numeric data as character data, supported by cfitsio, is *not* supported by CCfits.

Note the basic pattern of CCfits operations: they are performed on an object of type FITS. Access to HDU extension is provided by FITS:: member functions that return

references or pointers to objects representing HDUs. Extension are never created directly (all extension ctors are protected), but only through the functions `FITS::addTable` and `FITS::addImage` which add extensions to an existing FITS object, performing the necessary `cfitsio` calls.

The `FITS::addTable` function takes as one of its last arguments a HDU Type parameter, which needs to be `AsciiTbl` or `BinTbl`. The default is to create a `BinTable` (see next function).

Similarly, access to column data is provided through the functions `ExtHDU::Column`, which return references to columns specified by name or index number - see the documentation for the class `ExtHDU` for details.

`addTable` returns a pointer to `Table`, which is the abstract immediate superclass of the concrete classes `AsciiTable` and `BinTable`, whereas `addImage` returns a pointer to `ExtHDU`, which is the abstract base class of all FITS extensions. These base classes implement the public interface necessary to avoid the user of the library needing to downcast to a concrete type.

```
int writeAscii ()

//*****
// Create an ASCII Table extension containing 3 columns and 6 rows *
//*****
{
    // declare auto-pointer to FITS at function scope. Ensures no resources
    // leaked if something fails in dynamic allocation.
    std::auto_ptr<FITS> pFits(0);

    try
    {

        const std::string fileName("atestfil.fit");

        // append the new extension to file created in previous function call.

        // CCfits writing constructor.

        // if this had been a new file, then the following code would create
        // a dummy primary array with BITPIX=8 and NAXIS=0.

        pFits.reset( new FITS(fileName,Write) );
    }
    catch (CCfits::FITS::CantOpen)
    {
        // ... or not, as the case may be.
        return -1;
    }

    unsigned long rows(6);
    string hduName("PLANETS_ASCII");
    std::vector<string> colName(3,"");
    std::vector<string> colForm(3,"");
}
```



```

std::vector<string> colUnit(3,"");

/* define the name, datatype, and physical units for the 3 columns */
colName[0] = "Planet";
colName[1] = "Diameter";
colName[2] = "Density";

colForm[0] = "a8";
colForm[1] = "i6";
colForm[2] = "f4.2";

colUnit[0] = "";
colUnit[1] = "km";
colUnit[2] = "g/cm^-3";

std::vector<string> planets(rows);

const char *planet[] = {"Mercury", "Venus", "Earth",
                        "Mars", "Jupiter", "Saturn"};
const char *mnemoy[] = {"Many", "Volcanoes", "Erupt",
                        "Mulberry", "Jam", "Sandwiches", "Under",
                        "Normal", "Pressure"};

long diameter[] = { 4880,    12112,    12742,    6800,    143000,    121000}
;
float density[]  = { 5.1f,    5.3f,    5.52f,    3.94f,    1.33f,    0.69f}
;

// append a new ASCII table to the fits file. Note that the user
// cannot call the Ascii or Bin Table constructors directly as they
// are protected.

Table* newTable = pFits->addTable(hduName,rows,colName,colForm,colUnit,AsciiT
bl);
    size_t j = 0;
for ( ; j < rows; ++j) planets[j] = string(planet[j]);

// Table::column(const std::string& name) returns a reference to a Column obj
ect

try
{
    newTable->column(colName[0]).write(planets,1);
    newTable->column(colName[1]).write(diameter,rows,1);
    newTable->column(colName[2]).write(density,rows,1);

}
catch (FitsException&)
{
    // ExtHDU::column could in principle throw a NoSuchColumn exception,
    // or some other fits error may ensue.
    std::cerr << " Error in writing to columns - check e.g. that columns of
specified name "
                << " exist in the extension \n";

}

```

```

// FITSUtil::auto_array_ptr<T> is provided to counter resource leaks that
// may arise from C-arrays. It is a std::auto_ptr<T> analog that calls
// delete[] instead of delete.

FITSUtil::auto_array_ptr<long> pDiameter(new long[rows]);
FITSUtil::auto_array_ptr<float> pDensity(new float[rows]);
long* Cdiameter = pDiameter.get();
float* Cdensity = pDensity.get();

Cdiameter[0] = 4880; Cdiameter[1] = 12112; Cdiameter[2] = 12742; Cdiameter[3]
    = 6800;
Cdiameter[4] = 143000; Cdiameter[5] = 121000;

Cdensity[0] = 5.1f; Cdensity[1] = 5.3f; Cdensity[2] = 5.52f;
    Cdensity[3] = 3.94f; Cdensity[4] = 1.33f; Cdensity[5] = 0.69;

// this << operator outputs everything that has been read.

std::cout << *newTable << std::endl;

pFits->pHDU().addKey("NEWVALUE",42," Test of adding keyword to different exte
nsion");

    pFits->pHDU().addKey("STRING",std::string(" Rope "), "trailing blank test
1 ");

    pFits->pHDU().addKey("STRING2",std::string("Cord"), "trailing blank test 2
");
// demonstrate increaing number of rows and null values.
long ignoreVal(12112);
long nullNumber(-999);
try
{
    // add a TNULn value to column 2.
    newTable->column(colName[1]).addNullValue(nullNumber);
    // test that writing new data properly expands the number of rows
    // in both the file).write(planets,rows-3);
    newTable->column(colName[2]).write(density,rows,rows-3);
    // test the undefined value functionality. Undefineds are replaced on
    // disk but not in the memory copy.
    newTable->column(colName[1]).write(diameter,rows,rows-3,&ignoreVal);
}
catch (FitsException&)
{
    // this time we're going to ignore problems in these operations

}

// output header information to check that everything we did so far
// hasn't corrupted the file.

std::cout << pFits->pHDU() << std::endl;

std::vector<string> mnemon(9);
for ( j = 0; j < 9; ++j) mnemon[j] = string(mnemoy[j]);

```

```

// Add a new column of string type to the Table.
// type,  columnName, width, units. [optional - decimals, column number]
// decimals is only relevant for floatingpoint data in ascii columns.
newTable->addColumn(Tstring,"Mnemonic",10," words ");
newTable->column("Mnemonic").write(mnemon,1);

// write the data string.
newTable->writeDate();

// and see if it all worked right.
std::cout << *newTable << std::endl;

return 0;
}

```

## 8 Creating and Writing to a Binary Table Extension

The Binary Table interface is more complex because there is an additional parameter, the vector size of each ‘cell’ in the table, the need to support variable width columns, and the desirability of supporting the input of data in various formats.

The interface supports writing to vector tables the following data structures: C-arrays (T\*), std::vector<T> objects, std::valarray<T> objects, and std::vector<valarray<T> >. The last of these is the internal representation of the data.

The function below exercises the following functionality:

- Create a BinTable extension
- Write vector rows to the table
- Insert table rows
- Write complex data to both scalar and vector columns.
- Insert Table columns
- Delete Table rows
- Write HISTORY and COMMENT cards to the Table

```

int writeBinary ()

//*****
// Create a BINARY table extension and write and manipulate vector rows
//*****
{
    std::auto_ptr<FITS> pFits(0);
}

```

```

try
{
    const std::string fileName("atestfil.fit");
    pFits.reset( new FITS(fileName,Write) );
}
catch (CCfits::FITS::CantOpen)
{
    return -1;
}

unsigned long rows(3);
string hduName("TABLE_BINARY");
std::vector<string> colName(7,"");
std::vector<string> colForm(7,"");
std::vector<string> colUnit(7,"");

colName[0] = "numbers";
colName[1] = "sequences";
colName[2] = "powers";
colName[3] = "big-integers";
colName[4] = "dcomplex-roots";
colName[5] = "fcomplex-roots";
colName[6] = "scalar-complex";

colForm[0] = "8A";
colForm[1] = "20J";
colForm[2] = "20D";
colForm[3] = "20V";
colForm[4] = "20M";
colForm[5] = "20C";
colForm[6] = "1M";

colUnit[0] = "magnets";
colUnit[1] = "bulbs";
colUnit[2] = "batteries";
colUnit[3] = "mulberries";
colUnit[4] = "";
colUnit[5] = "";
colUnit[6] = "pico boo";

std::vector<string> numbers(rows);

const string num("NUMBER-");
for (size_t j = 0; j < rows; ++j)
{
#ifdef HAVE_STRSTREAM
    std::ostringstream pStr;
#else
    std::ostringstream pStr;
#endif
    pStr << num << j+1;
    numbers[j] = string(pStr.str());
}

```

```

const size_t OFFSET(20);

// write operations take in data as valarray<T>, vector<T> , and
// vector<valarray<T> >, and T* C-arrays. Create arrays to exercise the C++
// containers. Check complex I/O for both float and double complex types.

std::valarray<long> sequence(60);
std::vector<long> sequenceVector(60);
std::vector<std::valarray<long> > sequenceVV(3);

for (size_t j = 0; j < rows; ++j)
{
    sequence[OFFSET*j] = 1 + j;
    sequence[OFFSET*j+1] = 1 + j;
    sequenceVector[OFFSET*j] = sequence[OFFSET*j];
    sequenceVector[OFFSET*j+1] = sequence[OFFSET*j+1];
    // generate Fibonacci numbers.
    for (size_t i = 2; i < OFFSET; ++i)
    {
        size_t elt (OFFSET*j+i);
        sequence[elt] = sequence[elt-1] + sequence[elt - 2];
        sequenceVector[elt] = sequence[elt] ;
    }
    sequenceVV[j].resize(OFFSET);
    sequenceVV[j] = sequence[std::slice(OFFSET*j,OFFSET,1)];
}

std::valarray<unsigned long> unsignedData(60);
unsigned long base (1 << 31);
std::valarray<double> powers(60);
std::vector<double> powerVector(60);
std::vector<std::valarray<double> > powerVV(3);
std::valarray<std::complex<double> > croots(60);
std::valarray<std::complex<float> > fcroots(60);
std::vector<std::complex<float> > fcroots_vector(60);
std::vector<std::valarray<std::complex<float> > > fcrootv(3);

// create complex data as 60th roots of unity.
double PIBY = M_PI/30.;

for (size_t j = 0; j < rows; ++j)
{
    for (size_t i = 0; i < OFFSET; ++i)
    {
        size_t elt (OFFSET*j+i);
        unsignedData[elt] = sequence[elt];
        croots[elt] = std::complex<double>(std::cos(PIBY*elt), std::sin(PI
        BY*elt));
        fcroots[elt] = std::complex<float>(croots[elt].real(), croots[elt]
        .imag());
        double x = i+1;
        powers[elt] = pow(x, j+1);
        powerVector[elt] = powers[elt];
    }
}

```

```

    }
    powerVV[j].resize(OFFSET);
    powerVV[j] = powers[std::slice(OFFSET*j,OFFSET,1)];
}

FITSUtil::fill(fcroots_vector,fcroots[std::slice(0,20,1)]);

unsignedData += base;
// syntax identical to Binary Table

Table* newTable = pFits->addTable(hduName,rows,colName,colForm,colUnit);

// numbers is a scalar column

newTable->column(colName[0]).write(numbers,1);

// write valarrays to vector column: note signature change
newTable->column(colName[1]).write(sequence,rows,1);
newTable->column(colName[2]).write(powers,rows,1);
newTable->column(colName[3]).write(unsignedData,rows,1);
newTable->column(colName[4]).write(croots,rows,1);
newTable->column(colName[5]).write(fcroots,rows,3);
newTable->column(colName[6]).write(fcroots_vector,1);
// write vectors to column: note signature change

newTable->column(colName[1]).write(sequenceVector,rows,4);
newTable->column(colName[2]).write(powerVector,rows,4);

std::cout << *newTable << std::endl;

for (size_t j = 0; j < 3 ; ++j)
{
    fcrootv[j].resize(20);
    fcrootv[j] = fcroots[std::slice(20*j,20,1)];
}

// write vector<valarray> object to column.

newTable->column(colName[1]).writeArrays(sequenceVV,7);
newTable->column(colName[2]).writeArrays(powerVV,7);


// create a new vector column in the Table

newTable->addColumn(Tfloat,"powerSeq",20,"none");

// add data entries to it.

newTable->column("powerSeq").writeArrays(powerVV,1);
newTable->column("powerSeq").write(powerVector,rows,4);
newTable->column("dcomplex-roots").write(croots,rows,4);
newTable->column("powerSeq").write(sequenceVector,rows,7);

std::cout << *newTable << std::endl;

```

```
// delete one of the original columns.

newTable->deleteColumn(colName[2]);

// add a new set of rows starting after row 3. So we'll have 14 with
// rows 4,5,6,7,8 blank

newTable->insertRows(3,5);

// now, in the new column, write 3 rows (sequenceVV.size() = 3). This
// will place data in rows 3,4,5 of this column,overwriting them.

newTable->column("powerSeq").writeArrays(sequenceVV,3);
newTable->column("fcomplex-roots").writeArrays(fcrootv,3);

// delete 3 rows starting with row 2. A Table:: method, so the same
// code is called for all Table objects. We should now have 11 rows.

newTable->deleteRows(2,3);

//add a history string. This function call is in HDU:: so is identical
//for all HDUs

string hist("This file was created for testing CCfits write functionality");
hist += " it serves no other useful purpose. This particular part of the file
      was ";
hist += " constructed to test the writeHistory() and writeComment() functiona
      lity" ;

newTable->writeHistory(hist);

// add a comment string. Use std::string method to change the text in the mes
      sage
// and write the previous junk as a comment.

hist.insert(0, " COMMENT TEST ");

newTable->writeComment(hist);

// ... print the result.

std::cout << *newTable << std::endl;

return 0;
}
```

## 9 Copying an Extension between Files

Copying extensions from one fits file to another is very straightforward. A complication arises, however, because CCfits requires every FITS object to correspond to a conforming FITS file once constructed. Thus we provide a custom constructor which copies the primary HDU of a “source FITS file into a new file. Subsequent extensions can be copied by name or extension number as illustrated below.

Note that the simple call

```
FITS::FITS(const std::string& filename)
```

Reads the headers for all of the extensions in the file, so that after the FITS object corresponding to *infile* in the following code is instantiated, all extensions are recognized [read calls are also provided to read only specific HDUs - see below].

In the example code below, the file *outFile* is written straight to disk. Since the code never requests that the HDUs being written to that file are read, the user needs to add statements to do this after the copy is complete.

```
int copyHDU()
{
    //*****
    // copy the 1st and 3rd HDUs from the input file to a new FITS file
    //*****

    const string inFileName("atestfil.fit");
    const string outFileName("btestfil.fit");

    int status(0);

    status = 0;

    remove(outFileName.c_str());          // Delete old file if it already exists

    // open the existing FITS file
    FITS inFile(inFileName);

    // custom constructor FITS::FITS(const string&, const FITS&) for
    // this particular task.

    FITS outFile(outFileName,inFile);

    // copy extension by number...
    outFile.copy(inFile.extension(2));

    // copy extension by name...
    outFile.copy(inFile.extension("TABLE_BINARY"));

    return 0;
}
```



## 10 Selecting Table Data

This function demonstrates the operation of filtering a table by selecting rows that satisfy a condition and writing them to a new file, or overwriting a table with the filtered data. A third mode, where a filtered dataset is appended to the file containing the source data, will be available shortly, but is currently not supported by cfitsio.

The expression syntax for the conditions that may be applied to table data are described in the [cfitsio manual](#). In the example below, we illustrate filtering with a boolean expression involving one of the columns.

The two flags at the end of the call to FITS::filter are an 'overwrite' flag - which only has meaning if the inFile and outFile are the same, and a 'read' flag. overwrite defaults to true. The second flag is a 'read' flag which defaults to false. When set true the user has immediate access to the filtered data.

(Also see the section "Reading with Extended File Name Syntax")

```
int selectRows()
{
    const string inFile("atestfil.fit");
    const string outFile("btestfil.fit");
    const string newFile("ctestfil.fit");
    remove(newFile.c_str());

    // test 1: write to a new file
    std::auto_ptr<FITS> pInfile(new FITS(inFile,Write,string("PLANETS_ASCII"
))) ;
    FITS* infile(pInfile.get());
    std::auto_ptr<FITS> pNewfile(new FITS(newFile,Write));
    ExtHDU& source = infile->extension("PLANETS_ASCII");
    const string expression("DENSITY > 3.0");

    Table& sink1 = pNewfile->filter(expression,source,false,true);

    std::cout << sink1 << std::endl;

    // test 2: write a new HDU to the current file, overwrite false, read true.
    // AS OF 7/2/01 does not work because of a bug in cfitsio, but does not
    // crash, simply writes a new header to the file without also writing the
    // selected data.
    Table& sink2 = infile->filter(expression,source,false,true);

    std::cout << sink2 << std::endl;

    // reset the source file back to the extension in question.
    source = infile->extension("PLANETS_ASCII");

    // test 3: overwrite the current HDU with filtered data.
    Table& sink3 = infile->filter(expression,source,true,true);
```

```
std::cout << sink3 << std::endl;

return 0;
}
```

## 11 Reading Header information from a HDU

This function demonstrates selecting one HDU from the file, reading the header information and printing out the keys that have been read and the descriptions of the columns.

The `readData` flag is by default false (see below for the alternative case), which means that the data in the column is not read.

```
int readHeader()
{

    const string SPECTRUM("SPECTRUM");

    // read a particular HDU within the file. This call reads just the header
    // information from SPECTRUM

    std::auto_ptr<FITS> pInfile(new FITS("file1.pha", Read, SPECTRUM));

    // define a reference for clarity. (std::auto_ptr<T>::get returns a pointer

    ExtHDU& table = pInfile->extension(SPECTRUM);

    // read all the keywords, excluding those associated with columns.

    table.readAllKeys();

    // print the result.

    std::cout << table << std::endl;

    return 0;
}
```

## 12 Reading an Image

Image reading calls are made very simple: the FITS object is created with the `readDataFlag` set to true, and reading is done on construction. The following call

```
image.read(contents)
```

calls

```
PHDU::read(std::valarray<S>& image).
```

This copies the entire image from the FITS object into the `std::valarray` object contents, sizing it as necessary. `PHDU::read()` and `ExtHDU::read()` [for image extensions] take a range of arguments that support (a) reading the entire image - as in this example; (b) sections of an image starting from a given pixel; (c) rectangular subsets. See the class references for `PHDU` and `ExtHDU` for details.

```
int readImage()
{
    std::auto_ptr<FITS> pInfile(new FITS("atestfil.fit",Read,true));

    PHDU& image = pInfile->PHDU();

    std::valarray<unsigned long> contents;

    // read all user-specified, coordinate, and checksum keys in the image
    image.readAllKeys();

    image.read(contents);

    // this doesn't print the data, just header info.
    std::cout << image << std::endl;

    long ax1(image.axis(0));
    long ax2(image.axis(1));

    for (long j = 0; j < ax2; j+=10)
    {
        std::ostream_iterator<short> c(std::cout, "\t");
        std::copy(&contents[j*ax1], &contents[(j+1)*ax1-1], c);
        std::cout << '\n';
    }

    std::cout << std::endl;
    return 0;
}
```

## 13 Reading a Table Extension

Reading table data is similarly straightforward (unsurprisingly, because this application is exactly what `CCfits` was designed to do easily in the first place).

The two extensions are read on construction, including all the column data [`readDataFlag == true`] and then printed.

Note that if the data are read as part of the construction, then `CCfits` uses the row-optimization techniques describe in chapter 13 of the `cfitsio` manual; a chunk of data equal to the size of the available buffer space is read from contiguous disk blocks and

transferred to memory storage, as opposed to each column being read in turn. Thus the most efficient way of reading files is to acquire the data on construction.

```
int readTable()
{
    // read a table and explicitly read selected columns. To read instead all
    // the
    // data on construction, set the last argument of the FITS constructor
    // call to 'true'. This functionality was tested in the last release.
    std::vector<string> hdus(2);
    hdus[0] = "PLANETS_ASCII";
    hdus[1] = "TABLE_BINARY";

    std::auto_ptr<FITS> pInfile(new FITS("atestfil.fit",Read,hdus,false));

    ExtHDU& table = pInfile->extension(hdus[1]);

    std::vector < valarray <int > > pp;
    table.column("powerSeq").readArrays( pp, 1,3 );

    std::vector < valarray <std::complex<double> > > cc;
    table.column("dcomplex-roots").readArrays( cc, 1,3 );

    std::valarray < std::complex<float> > ff;
    table.column("fcomplex-roots").read( ff, 4 );

    std::cout << pInfile->extension(hdus[0]) << std::endl;

    std::cout << pInfile->extension(hdus[1]) << std::endl;

    return 0;
}
```

## 14 Reading with Extended File Name Syntax

It is also possible to apply extended file name syntax (as described in chapter 10 of the [cfitsio manual](#)) when reading data. The function below shows a typical example using the basic [CCfits::FITS](#) constructor.

The extended syntax is entered as part of the file name string. In this case it specifies an HDU and a row selection criterion dependent upon the values in the column named "Density." Any read operations performed on this HDU will only see rows which meet the "Density > 5.2" condition. Also the current header position in the file is automatically placed at the specified HDU upon construction of the FITS object.

Extended file name syntax can also be used with the FITS constructors which take specific HDU names or indices as arguments. However if the extended syntax specifies an HDU, that HDU must also be among those specified as a FITS constructor argument, otherwise a [CCfits::FITS::OperationNotSupported](#) exception is thrown. For example:

```
FITS fits(new FITS("myFile.fit[HDU_A]", Read, string("HDU_A"))); // OK
FITS fits(new FITS("myFile.fit[HDU_B]", Read, string("HDU_A"))); // Error
```

(Note - The extended file name feature which allows the opening of a particular image located in the row of a table remains unsupported in CCfits.)

```
int readExtendedSyntax()
{
    // Current extension will be set to PLANETS_ASCII after construction:
    std::auto_ptr<FITS> pInfile(new FITS("btestfil.fit[PLANETS_ASCII][Density > 5
    .2]"));
    std::cout << "\nCurrent extension: "
              << pInfile->currentExtensionName() << std::endl;

    Column& col = pInfile->currentExtension().column("Density");
    std::vector<double> densities;

    // nRows should only include rows with density column vals > 5.2.
    const int nRows = col.rows();
    col.read(densities, 1, nRows);
    for (int i=0; i<nRows; ++i)
        std::cout << densities[i] << " ";
    std::cout << std::endl;

    return 0;
}
```

## 15 What's Present, What's Missing, and Calling CFITSIO

Most of the functionality of cfitsio described in Chapter 5 of the cfitsio manual is present, although CCfits is designed to provide atomic read/write operations rather than primitive file manipulation. For example, opening and creating FITS files are private operations which are called by reading and writing constructors. Similarly, errors are treated by C++ exception handling rather than returning status codes, and moving between HDUs within a file is a primitive rather than an atomic operation [in CCfits, operations typically call an internal HDU::makeThisCurrent() call on a specific table or image extension, and then perform the requested read/write operation].

Read/Write operations for keys (in the HDU class) are provided; these implement calls to fits\_read\_key and fits\_update\_key respectively. In the case of keywords, which have one of five data types (Integer, Logical, String, Floating and Complex) CCfits will handle certain type conversions between the keyword value and the data type of the user-supplied variable. This is described in detail in the Keyword class reference page. In reading image and table data, intrinsic type conversions are performed as in cfitsio with the exception that reading numeric data into character data is **not** supported. There is an extensive set of member functions supporting equivalents of most of cfitsio's read/write operations: the classes PHDU [primary HDU] and ExtHDU [with

subclasses template `<typename T> ImageExt<T>`], provide multiple overloaded versions of read and write functions. The `Column` class, instances of which can be held in `Table` instances [with subclasses `AsciiTable` and `BinTable`] has also an extensive set of read/write operations.

A special constructor is provided which creates a new file with the Primary HDU of a source file. A `FITS::copy(const HDU&)` function copies HDUs from one file into another. Support for filtering table rows by expression is provided by a `FITS::filter( ... )` call which may be used to create a new filtered file or overwrite an existing HDU (see `cfitsio` manual section 5.6.4).

Functions are provided for adding and deleting columns, and inserting and deleting rows in tables.

HDU objects also have functions to implement writing of history, comment and date keys.

Extended file name syntax (chapter 10 of the `cfitsio` manual) is supported in general, though not the feature which allows the opening of a particular image stored in the row of a table.

## 15.1 What's Not Present

The coordinate library manipulations [`cfitsio` manual chapter 7] are not supported.

The iterator work functions [`cfitsio` manual chapter 6] are not supported. Many of the functions provided are easier to implement using the properties of the standard library, since the standard library containers either allow vectorized operations (in the case of `valarrays`) or standard library algorithms that take iterators or pointers. In some ways the `fits_iterate_data` function provide an alternative, approach to the same need for encapsulation addressed by `CCfits`.

The hierarchical grouping routines are not supported.

Explicit opening of in-memory data sets as described in Chapter 9 of the manual is *not* supported since none of the `FITS` constructors call the appropriate `cfitsio` primitives.

## 15.2 Calling CFITSIO

To gain any functionality currently missing in `CCfits`, it is possible of course to call the underlying `CFITSIO` library functions directly. The `CCfits` `FITS` and `HDU` classes both have the public member function `fitsPointer()`, which returns the `fitsfile` pointer required in `CFITSIO` function calls. One should use caution when doing this however, since any I/O changes made this way will NOT be mirrored in the `CCfits` `FITS` object (nor its component objects) associated with the file. Therefore once a `FITS` object has been bypassed this way, it is safest to just not use that object again, and instead continue calling `CFITSIO` directly or instantiate a new `FITS` object that will pick up the current file state.

## 16 Previous Release Notes

Changes for CCfits 2.0 release Feb 2008 Enhancements to CCfits:

- **Checksum Capability:** 4 checksum related functions have been added to the HDU class, which now allows users to set and verify checksums directly from inside CCfits.
- **Capturing Error Messages:** The FitsException base class now stores its output error message, and it can be retrieved from any of the exception subclass objects with a call to the new FitsException::message() function.
- **Improved Keyword Handling:** New functions copyAllKeys, keywordCategories, and a second addKey function have been added to the HDU class. The Keyword class now offers a public setValue function to modify an existing keyword. Also the class member documentation for keyword related functions has been upgraded and expanded.
- **Image Scaling:** In the HDU class (for instances of its PHDU and image ExtHDU subclasses), scale and zero set functions can now write BSCALE and BZERO keywords to the file. A new suppressScaling function has been added to temporarily turn off scaling. The ImageExt<T> class has also been added to the documentation.
- **Miscellaneous New Functions:** Table::getRowsize() (submitted by Patrik Jonsson), Fits::fitsPointer(), Column::parent().

Bug Fixes:

- FITS constructor in Write mode caused a segmentation fault when used on read-only files. (Reported by Gerard Zins)
- Column write functions were not turning off NULL checking even when the nulval pointer was set to 0. (Reported by Gerard Zins)
- For the FITS constructor which takes an existing FITS object as an argument, when given the filename of an existing file (and without the '!' specifier), it places a new primary HDU in the first extension. It shouldn't allow a write operation at all in this case. (Reported by Andy Beardmore)
- Some additional include statements are needed for compilation on a test version of g++4.3 (Reported by Aurelien Jarno)

Backwards Compatibility Issues:

- The following documented public access member functions have now been removed or made protected/private. As these functions were either never

fully implemented or could not successfully be used from external code, it is hoped that these removals will not break any pre-existing code: `FITS::clone`, `HDU::setKeyword`, the `HDU::bitpix` set function, the `Keyword` class constructors.

Changes for CCfits 1.8 release 10/07.

- Fixes made to bugs in `Column` write and `writeArrays` functions which were preventing the writing of variable-width columns. Also now allows writing to fixed-width columns with arrays that are shorter than the fixed width.
- The `HDU::readAllKeys()` function will no longer throw if it is unable to read a particular keyword. Instead it will skip it and move to the next keyword. This was done primarily to prevent it from tripping on undefined keywords.

Changes for CCfits 1.7 release 6/07. Fixes for the following bugs:

- The `FITS::copy` function merely wrote the copied HDU to the file, but did not allow it to be accessed for further modifications within CCfits.
- When reading compressed images, CCfits should use the `ZBITPIX` and `ZNAXIS` keywords rather than `BITPIX` and `NAXIS`. (Fix is based on a patch submitted by Patrik Jonsson.)
- The `BSCALE` keyword was being ignored if the `BZERO` keyword didn't also exist.
- Cases of out-of-scope usage of `std::string`'s `c_str()` pointers, could potentially cause crash. (Fix submitted by Jeremy Sanders.)

Changes for CCfits 1.6 release 11/06

- Added capability to write compressed images, including 6 new wrapper public functions in `FITS` class.
- In `FITS::addImage`, corrected the logic which checks for a pre-existing image extension with the same version number.
- CFITSIO 3.02 renamed `fitsfile` struct member `rice_nbits` to `noise_nbits`. Made corresponding change in `copyFitsPtr` function in `FITSUtil.cxx`. As it stands, this makes this version of CCfits incompatible with earlier versions of CFITSIO
- In `FITS.h` definition, removed both friend declarations of `HDUCreator` Make functions. It seems neither function needs to be a friend, and one of them is actually private. Some compilers don't allow this (report came from MS VisualC++ user).



- Bug fix in Make function of HDUCreator.cxx. When creating a new ImageExt (and not the primary), it was only passing the version number along for float and double types. This causes problems when there is more than 1 image extension with the same name, and it needs the version number to distinguish them.
- A couple of bug fixes to the first/last/stride version of PHDU read image subset. It was not passing the proper parameters to fits\_read\_subset, and was not always correctly resizing the internal m\_image array.

## 17 Todo List

Member **CCfits::AsciiTable::AsciiTable**(FITSBase \*p, const String &hduName, int rows, const std::vector< String > &colNames, int version=1)  
 {enforce equal dimensions for arrays input to AsciiTable, BinTable writing ctor}

Member **CCfits::FITS::addImage**(const String &hduName, int bpix, std::vector< long > &naxes, int version=1)  
 Add a function for replacing the primary image

Member **CCfits::FITS::addTable**(const String &hduName, int rows, const std::vector< String > &columnNames, int version=1)  
 the code should one day check that the version keyword is higher than any other versions already added to the FITS object (although cfitsio doesn't do this either).

Class **CCfits::PHDU** Implement functions that allow replacement of the primary image

## 18 Module Index

### 18.1 Modules

Here is a list of all modules:

FITS Exceptions

36

## 19 Namespace Index

### 19.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

**FITSUtil** (**FITSUtil** is a namespace containing functions used internally by CCfits, but which might be of use for other applications ) 38

## 20 Hierarchical Index

### 20.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

<b>CCfits::Column</b>	<b>50</b>
<b>CCfits::FITS</b>	<b>79</b>
<b>CCfits::FitsException</b>	<b>95</b>
<b>CCfits::Column::InsufficientElements</b>	<b>111</b>
<b>CCfits::Column::InvalidDataType</b>	<b>112</b>
<b>CCfits::Column::InvalidNumberOfRows</b>	<b>115</b>
<b>CCfits::Column::InvalidRowNumber</b>	<b>116</b>
<b>CCfits::Column::InvalidRowParameter</b>	<b>117</b>
<b>CCfits::Column::NoNullValue</b>	<b>125</b>
<b>CCfits::Column::RangeError</b>	<b>138</b>
<b>CCfits::Column::WrongColumnType</b>	<b>146</b>
<b>CCfits::ExtHDU::WrongExtensionType</b>	<b>147</b>
<b>CCfits::FITS::CantCreate</b>	<b>48</b>
<b>CCfits::FITS::CantOpen</b>	<b>49</b>
<b>CCfits::FITS::NoSuchHDU</b>	<b>128</b>
<b>CCfits::FITS::OperationNotSupported</b>	<b>130</b>
<b>CCfits::FitsError</b>	<b>94</b>
<b>CCfits::FITSUtil::UnrecognizedType</b>	<b>146</b>
<b>CCfits::HDU::InvalidExtensionType</b>	<b>113</b>

CCfits::HDU::InvalidImageDataType	114
CCfits::HDU::NoNullValue	124
CCfits::HDU::NoSuchKeyword	129
CCfits::Table::NoSuchColumn	126
CCfits::FitsFatal	97
CCfits::FITSUtil::auto_array_ptr< X >	42
CCfits::FITSUtil::CAarray< T >	47
CCfits::FITSUtil::CVAarray< T >	67
CCfits::FITSUtil::CVarray< T >	68
CCfits::FITSUtil::MatchName< T >	121
CCfits::FITSUtil::MatchNum< T >	122
CCfits::FITSUtil::MatchPtrName< T >	123
CCfits::FITSUtil::MatchPtrNum< T >	123
CCfits::FITSUtil::MatchType< T >	123
CCfits::HDU	98
CCfits::ExtHDU	68
CCfits::ImageExt< T >	108
CCfits::Table	139
CCfits::AsciiTable	38
CCfits::BinTable	44
CCfits::PHDU	131
CCfits::Keyword	118

## 21 Class Index

### 21.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<b>CCfits::AsciiTable</b> (Class Representing Ascii Table Extensions )	<b>38</b>
<b>CCfits::BinTable</b> (Class Representing Binary Table Extensions. Contains columns with scalar or vector row entries )	<b>44</b>
<b>CCfits::Column</b> (Abstract base class for Column objects )	<b>50</b>
<b>CCfits::Column::InsufficientElements</b> (Exception thrown if the data supplied for a write operation is less than declared )	<b>111</b>
<b>CCfits::Column::InvalidDataType</b> (Exception thrown for invalid data type inputs )	<b>112</b>
<b>CCfits::Column::InvalidNumberOfRows</b> (Exception thrown if user enters a non-positive number for the number of rows to write )	<b>115</b>
<b>CCfits::Column::InvalidRowNumber</b> (Exception thrown on attempting to read a row number beyond the end of a table )	<b>116</b>
<b>CCfits::Column::InvalidRowParameter</b> (Exception thrown on incorrect row writing request )	<b>117</b>
<b>CCfits::Column::NoNullValue</b> (Exception thrown if a null value is specified without support from existing column header )	<b>125</b>
<b>CCfits::Column::RangeError</b> (Exception to be thrown for inputs that cause range errors in column read operations )	<b>138</b>
<b>CCfits::Column::WrongColumnType</b> (Exception thrown on attempting to access a scalar column as vector data )	<b>146</b>
<b>CCfits::ExtHDU</b> (Base class for all FITS extension HDUs, i.e. Image Extensions and Tables )	<b>68</b>
<b>CCfits::ExtHDU::WrongExtensionType</b> (Exception to be thrown on unmatched extension types )	<b>147</b>
<b>CCfits::FITS</b> (Memory object representation of a disk FITS file )	<b>79</b>
<b>CCfits::FITS::CantCreate</b> (Thrown on failure to create new file )	<b>48</b>

<a href="#">CCfits::FITS::CantOpen</a> (Thrown on failure to open existing file )	49
<a href="#">CCfits::FITS::NoSuchHDU</a> (Exception thrown by <a href="#">HDU</a> retrieval methods )	128
<a href="#">CCfits::FITS::OperationNotSupported</a> (Thrown for unsupported operations, such as attempted to select rows from an image extension )	130
<a href="#">CCfits::FitsError</a> ( <a href="#">FitsError</a> is the exception thrown by non-zero cfitsio status codes )	94
<a href="#">CCfits::FitsException</a> ( <a href="#">FitsException</a> is the base class for all exceptions thrown by this library )	95
<a href="#">CCfits::FitsFatal</a> ([potential] base class for exceptions to be thrown on internal library error )	97
<a href="#">CCfits::FITSUtil::auto_array_ptr&lt; X &gt;</a> (A class that mimics the std:: library auto_ptr class, but works with arrays )	42
<a href="#">CCfits::FITSUtil::CAarray&lt; T &gt;</a> (Function object returning C array from a valarray. see <a href="#">CVarray</a> for details )	47
<a href="#">CCfits::FITSUtil::CVAarray&lt; T &gt;</a> (Function object returning C array from a vector of valarrays. see <a href="#">CVarray</a> for details )	67
<a href="#">CCfits::FITSUtil::CVarray&lt; T &gt;</a> (Function object class for returning C arrays from standard library objects used in the <a href="#">FITS</a> library implementation )	68
<a href="#">CCfits::FITSUtil::MatchName&lt; T &gt;</a> (Predicate for classes that have a name attribute; match input string with instance name )	121
<a href="#">CCfits::FITSUtil::MatchNum&lt; T &gt;</a> (Predicate for classes that have an index attribute; match input index with instance value )	122
<a href="#">CCfits::FITSUtil::MatchPtrName&lt; T &gt;</a> (As for <a href="#">MatchName</a> , only with the input class a pointer )	123
<a href="#">CCfits::FITSUtil::MatchPtrNum&lt; T &gt;</a> (As for <a href="#">MatchNum</a> , only with the input class a pointer )	123
<a href="#">CCfits::FITSUtil::MatchType&lt; T &gt;</a> (Function object that returns the <a href="#">FITS</a> ValueType corresponding to an input intrinsic type )	123
<a href="#">CCfits::FITSUtil::UnrecognizedType</a> (Exception thrown by <a href="#">MatchType</a> if it encounters data type incompatible with cfitsio )	146

<a href="#">CCfits::HDU</a> (Base class for all <a href="#">HDU</a> [Header-Data Unit] objects )	98
<a href="#">CCfits::HDU::InvalidExtensionType</a> (Exception to be thrown if user requests extension type that can not be understood as <a href="#">ImageExt</a> , <a href="#">AsciiTable</a> or <a href="#">BinTable</a> )	113
<a href="#">CCfits::HDU::InvalidImageDataType</a> (Exception to be thrown if user requests creation of an image of type not supported by cfitsio )	114
<a href="#">CCfits::HDU::NoNullValue</a> (Exception to be thrown on seek errors for keywords )	124
<a href="#">CCfits::HDU::NoSuchKeyword</a> (Exception to be thrown on seek errors for keywords )	129
<a href="#">CCfits::ImageExt&lt; T &gt;</a>	108
<a href="#">CCfits::Keyword</a> (Abstract base class defining the interface for <a href="#">Keyword</a> objects )	118
<a href="#">CCfits::PHDU</a> (Class representing the primary <a href="#">HDU</a> for a <a href="#">FITS</a> file )	131
<a href="#">CCfits::Table</a>	139
<a href="#">CCfits::Table::NoSuchColumn</a> (Exception to be thrown on a failure to retrieve a column specified either by name or index number )	126

## 22 Module Documentation

### 22.1 FITS Exceptions

#### Classes

- class [CCfits::Column::RangeError](#)  
*exception to be thrown for inputs that cause range errors in column read operations.*
- class [CCfits::ExtHDU::WrongExtensionType](#)  
*Exception to be thrown on unmatched extension types.*
- class [CCfits::FITS::CantCreate](#)  
*thrown on failure to create new file*
- class [CCfits::FITS::CantOpen](#)  
*thrown on failure to open existing file*

- class `CCfits::FITS::NoSuchHDU`  
*exception thrown by `HDU` retrieval methods.*
- class `CCfits::FITS::OperationNotSupported`  
*thrown for unsupported operations, such as attempted to select rows from an image extension.*
- class `CCfits::FitsError`  
*`FitsError` is the exception thrown by non-zero `cfitsio` status codes.*
- class `CCfits::FitsException`  
*`FitsException` is the base class for all exceptions thrown by this library.*
- class `CCfits::FitsFatal`  
*[potential] base class for exceptions to be thrown on internal library error.*
- class `CCfits::FITSUtil::UnrecognizedType`  
*exception thrown by `MatchType` if it encounters data type incompatible with `cfitsio`.*
- class `CCfits::HDU::InvalidExtensionType`  
*exception to be thrown if user requests extension type that can not be understood as `ImageExt`, `AsciiTable` or `BinTable`.*
- class `CCfits::HDU::InvalidImageDataType`  
*exception to be thrown if user requests creation of an image of type not supported by `cfitsio`.*
- class `CCfits::HDU::NoNullValue`  
*exception to be thrown on seek errors for keywords.*
- class `CCfits::HDU::NoSuchKeyword`  
*exception to be thrown on seek errors for keywords.*
- class `CCfits::Table::NoSuchColumn`  
*Exception to be thrown on a failure to retrieve a column specified either by name or index number.*

## 23 Namespace Documentation

### 23.1 FITSUtil Namespace Reference

[FITSUtil](#) is a namespace containing functions used internally by CCfits, but which might be of use for other applications.

#### 23.1.1 Detailed Description

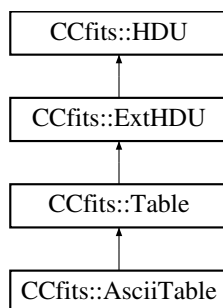
[FITSUtil](#) is a namespace containing functions used internally by CCfits, but which might be of use for other applications.

## 24 Class Documentation

### 24.1 CCfits::AsciiTable Class Reference

Class Representing Ascii [Table](#) Extensions.

`#include <AsciiTable.h>`Inheritance diagram for CCfits::AsciiTable::



#### Public Member Functions

- virtual void [addColumn](#) (ValueType type, const String &columnName, long repeatWidth, const String &colUnit=String(""), long decimals=0, size\_t columnNumber=0)

*add a new column to an existing table [HDU](#).*

- virtual [AsciiTable](#) \* [clone](#) (FITSBase \*p) const

*virtual copy constructor*



- virtual void [readData](#) (bool readFlag=false, const std::vector< String > &keys=std::vector< String >())

*read columns and keys specified in the input array.*

### Protected Member Functions

- [AsciiTable](#) (FITSBase \*p, int number)  
*read [AsciiTable](#) with [HDU](#) number `number` from existing file.*
- [AsciiTable](#) (FITSBase \*p, const String &hduName, int rows, const std::vector< String > &columnName=std::vector< String >(), const std::vector< String > &columnFmt=std::vector< String >(), const std::vector< String > &columnUnit=std::vector< String >(), int version=1)

*writing constructor: create new [Ascii Table](#) object with the specified columns*

- [AsciiTable](#) (FITSBase \*p, const String &hduName=String(""), bool readFlag=false, const std::vector< String > &keys=std::vector< String >(), int version=1)

*reading constructor: Construct a [AsciiTable](#) extension from an extension of an existing disk file.*

- [~AsciiTable](#) ()

*destructor.*

#### 24.1.1 Detailed Description

Class Representing [Ascii Table](#) Extensions. May only contain columns with scalar row entries and a small range of data types. [AsciiTable](#) (re)implements functions prescribed in the [Table](#) abstract class. The implementations allow the calling of cfitsio specialized routines for [AsciiTable](#) header construction.

Direct instantiation of [AsciiTable](#) objects is disallowed: they are created by explicit calls to [FITS::addTable](#)( ... ), [FITS::read](#)(...) or internally by one of the [FITS](#) ctors on initialization. The default for [FITS::addTable](#) is to produce [BinTable](#) extensions.

### 24.1.2 Constructor & Destructor Documentation

**24.1.2.1 CCfits::AsciiTable::AsciiTable (FITSBase \* *p*, const String & *hduName* = String(""), bool *readFlag* = false, const std::vector<String> & *keys* = std::vector<String>(), int *version* = 1) [protected]**

reading constructor: Construct a [AsciiTable](#) extension from an extension of an existing disk file. The [Table](#) is specified by name and optional version number within the file. An array of strings representing columns or keys indicates which data are to be read. The column data are only read if *readFlag* is true. Reading on construction is optimized, so it is more efficient to read data at the point of instantiation. This favours a "resource acquisition is initialization" model of data management.

#### Parameters:

- p* pointer to FITSBase object for internal use
- hduName* name of [AsciiTable](#) object to be read.
- readFlag* flag to determine whether to read data on construction
- keys* (optional) a list of keywords/columns to be read. The implementation will determine which are keywords. If none are specified, the constructor will simply read the header
- version* (optional) version number. If not specified, will read the first extension that matches *hduName*.

**24.1.2.2 CCfits::AsciiTable::AsciiTable (FITSBase \* *p*, const String & *hduName*, int *rows*, const std::vector<String> & *columnName* = std::vector<String>(), const std::vector<String> & *columnFmt* = std::vector<String>(), const std::vector<String> & *columnUnit* = std::vector<String>(), int *version* = 1) [protected]**

writing constructor: create new [Ascii Table](#) object with the specified columns The constructor creates a valid [HDU](#) which is ready for [Column::write](#) or [insertRows](#) operations. The disk [FITS](#) file is update accordingly. The data type of each column is determined by the *columnFmt* argument (TFORM keywords). See [cfitsio](#) documentation for acceptable values.

#### Parameters:

- hduName* name of [AsciiTable](#) object to be written

**rows** number of rows in the table (NAXIS2)  
**columnName** array of column names for columns to be constructed.  
**columnFmt** array of column formats for columns to be constructed.  
**columnUnit** (optional) array of units for data in columns.  
**version** (optional) version number for [HDU](#).

The dimensions of `columnType`, `columnName` and `columnFmt` must match, although this is not enforced at present.

#### Todo

{enforce equal dimensions for arrays input to [AsciiTable](#), [BinTable](#) writing ctor}

#### 24.1.2.3 CCfits::AsciiTable::AsciiTable (FITSBase \* *p*, int *number*) [protected]

read [AsciiTable](#) with [HDU](#) number `number` from existing file. This is used internally by methods that need to access HDUs for which no EXTNAME [or equivalent] keyword exists.

#### 24.1.3 Member Function Documentation

##### 24.1.3.1 void CCfits::AsciiTable::addColumn (ValueType *type*, const String & *columnName*, long *repeatWidth*, const String & *colUnit* = String(""), long *decimals* = 0, size\_t *columnNumber* = 0) [virtual]

add a new column to an existing table [HDU](#).

#### Parameters:

**type** The data type of the column to be added  
**columnName** The name of the column to be added  
**repeatWidth** for a string valued, this is the width of a string. For a numeric column it supplies the vector length of the rows. It is ignored for ascii table numeric data.  
**colUnit** an optional field specifying the units of the data (TUNITn)  
**decimals** optional parameter specifying the number of decimals for an ascii numeric column

*columnNumber* optional parameter specifying column number to be created. If not specified the column is added to the end. If specified, the column is inserted and the columns already read are reindexed. This parameter is provided as a convenience to support existing code rather than recommended.

Reimplemented from [CCfits::ExtHDU](#).

**24.1.3.2** `void CCfits::AsciiTable::readData (bool readFlag = false, const std::vector< String > & keys = std::vector<String>()) [virtual]`

read columns and keys specified in the input array. See [Table](#) class documentation for further details.

Implements [CCfits::ExtHDU](#).

The documentation for this class was generated from the following files:

- AsciiTable.h
- AsciiTable.cxx

## 24.2 CCfits::FITSUtil::auto\_array\_ptr< X > Class Template Reference

A class that mimics the std:: library auto\_ptr class, but works with arrays.

```
#include <FITSUtil.h>
```

### Public Member Functions

- [auto\\_array\\_ptr](#) (auto\_array\_ptr< X > &right) throw ()  
*copy constructor*
- [auto\\_array\\_ptr](#) (X \*p=0) throw ()  
*constructor. allows creation of pointer to null, can be modified by [reset\(\)](#)*
- [~auto\\_array\\_ptr](#) ()  
*destructor.*
- X \* [get](#) () const  
*return a token for the underlying content of \*this*

- X & **operator\*** () throw ()  
*deference operator*
- void **operator=** (auto\_array\_ptr< X > &right)  
*assignment operator: transfer of ownership semantics*
- X **operator[ ]** (size\_t i) const throw ()  
*return a copy of the ith element of the array*
- X & **operator[ ]** (size\_t i) throw ()  
*return a reference to the ith element of the array*
- X \* **release** () throw ()  
*return underlying content of \*this, transferring memory ownership*
- X \* **reset** (X \*p) throw ()  
*change the content of the auto\_array\_ptr to p*

### Static Public Member Functions

- static void **remove** (X \*&x)  
*utility function to delete the memory owned by x and set it to null.*

#### 24.2.1 Detailed Description

**template<typename X> class CCfits::FITSUtil::auto\_array\_ptr< X >**

A class that mimics the std:: library auto\_ptr class, but works with arrays. This code was written by Jack Reeves and first appeared C++ Report, March 1996 edition. Although some authors think one shouldn't need such a contrivance, there seems to be a need for it when wrapping C code.

Usage: replace

```
float* f = new float[200];
```

with

```
FITSUtil::auto_array_ptr<float> f(new float[200]);
```

Then the memory will be managed correctly in the presence of exceptions, and delete will be called automatically for f when leaving scope.

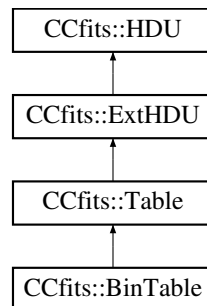
The documentation for this class was generated from the following file:

- FITSUtil.h

## 24.3 CCfits::BinTable Class Reference

Class Representing Binary [Table](#) Extensions. Contains columns with scalar or vector row entries.

#include <BinTable.h> Inheritance diagram for CCfits::BinTable::



### Public Member Functions

- virtual void [addColumn](#) (ValueType type, const String &columnName, long repeatWidth, const String &colUnit=String(""), long decimals=0, size\_t columnNumber=0)  
*add a new column to an existing table [HDU](#).*
- virtual [BinTable](#) \* [clone](#) (FITSBase \*p) const  
*virtual copy constructor*
- virtual void [readData](#) (bool readFlag=false, const std::vector< String > &keys=std::vector< String >())  
*read columns and keys specified in the input array.*

### Protected Member Functions

- [BinTable](#) (FITSBase \*p, int number)  
*read [BinTable](#) with [HDU](#) number `number` from existing file represented by fitsfile pointer `p`.*

- [BinTable](#) (FITSBase \*p, const String &hduName, int rows, const std::vector< String > &columnName=std::vector< String >(), const std::vector< String > &columnFmt=std::vector< String >(), const std::vector< String > &columnUnit=std::vector< String >(), int version=1)

*writing constructor*

- [BinTable](#) (FITSBase \*p, const String &hduName=String(""), bool readFlag=false, const std::vector< String > &keys=std::vector< String >(), int version=1)

*reading constructor.*

- [~BinTable](#) ()

*destructor.*

### 24.3.1 Detailed Description

Class Representing Binary [Table](#) Extensions. Contains columns with scalar or vector row entries. [BinTable](#) (re)implements functions prescribed in the [Table](#) abstract class. The implementations allow the calling of cfitsio specialized routines for [BinTable](#) header construction. functions particular to the [BinTable](#) class include those dealing with variable width columns

Direct instantiation of [BinTable](#) objects is disallowed: they are created by explicit calls to [FITS::addTable](#)( ... ), [FITS::read](#)(...) or internally by one of the [FITS](#) ctors on initialization. For addTable, creation of BinTables is the default.

### 24.3.2 Constructor & Destructor Documentation

#### 24.3.2.1 CCfits::BinTable::BinTable (FITSBase \*p, const String & hduName = String(""), bool readFlag = false, const std::vector< String > & keys = std::vector<String>(), int version = 1) [protected]

reading constructor. Construct a [BinTable](#) extension from an extension of an existing disk file. The [Table](#) is specified by name and optional version number within the file. An array of strings representing columns or keys indicates which data are to be read. The column data are only read if readFlag is true. Reading on construction is optimized, so it is more efficient to read data at the point of instantiation. This favours a "resource acquisition is initialization" model of data management.

#### Parameters:

*p* Pointer to FITSBase class, an internal implementation detail

**hduName** name of [BinTable](#) object to be read.

**readFlag** flag to determine whether to read data on construction

**keys** (optional) a list of keywords/columns to be read. The implementation will determine which are keywords. If none are specified, the constructor will simply read the header

**version** (optional) version number. If not specified, will read the first extension that matches hduName.

**24.3.2.2 CCfits::BinTable::BinTable (FITSBase \* *p*, const String & *hduName*, int *rows*, const std::vector< String > & *columnName* = std::vector<String>(), const std::vector< String > & *columnFmt* = std::vector<String>(), const std::vector< String > & *columnUnit* = std::vector<String>(), int *version* = 1) [protected]**

writing constructor The constructor creates a valid [HDU](#) which is ready for [Column::write](#) or [insertRows](#) operations. The disk [FITS](#) file is update accordingly. The data type of each column is determined by the *columnFmt* argument (TFORM keywords). See [cfitsio](#) documentation for acceptable values.

#### Parameters:

***p*** Pointer to [FITSBase](#) class, an internal implementation detail

***hduName*** name of [BinTable](#) object to be written

***rows*** number of rows in the table (NAXIS2)

***columnName*** array of column names for columns to be constructed.

***columnFmt*** array of column formats for columns to be constructed.

***columnUnit*** (optional) array of units for data in columns.

***version*** (optional) version number for [HDU](#).

The dimensions of *columnType*, *columnName* and *columnFmt* must match, but this is not enforced.

### 24.3.3 Member Function Documentation

**24.3.3.1 void CCfits::BinTable::addColumn (ValueType *type*, const String & *columnName*, long *repeatWidth*, const String & *colUnit* = String(""), long *decimals* = 0, size\_t *columnNumber* = 0) [virtual]**



add a new column to an existing table [HDU](#).

**Parameters:**

- type* The data type of the column to be added
- columnName* The name of the column to be added
- repeatWidth* for a string valued, this is the width of a string. For a numeric column it supplies the vector length of the rows. It is ignored for ascii table numeric data.
- colUnit* an optional field specifying the units of the data (TUNITn)
- decimals* optional parameter specifying the number of decimals for an ascii numeric column
- columnNumber* optional parameter specifying column number to be created. If not specified the column is added to the end. If specified, the column is inserted and the columns already read are reindexed. This parameter is provided as a convenience to support existing code rather than recommended.

Reimplemented from [CCfits::ExtHDU](#).

**24.3.3.2** `void CCfits::BinTable::readData (bool readFlag = false, const std::vector< String > & keys = std::vector<String> ()) [virtual]`

read columns and keys specified in the input array. See [Table](#) class documentation for further details.

Implements [CCfits::ExtHDU](#).

The documentation for this class was generated from the following files:

- BinTable.h
- BinTable.cxx

## 24.4 CCfits::FITSUtil::CAarray< T > Class Template Reference

function object returning C array from a valarray. see [CVarray](#) for details

```
#include <FITSUtil.h>
```

### Public Member Functions

- `T * operator() (const std::valarray< T > &inArray)`  
*operator returning C array for use with image data.*

### 24.4.1 Detailed Description

`template<typename T> class CCfits::FITSUtil::CAarray< T >`

function object returning C array from a valarray. see [CVarray](#) for details

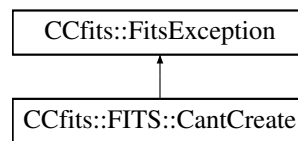
The documentation for this class was generated from the following file:

- FITSUtil.h

## 24.5 CCfits::FITS::CantCreate Class Reference

thrown on failure to create new file

`#include <FITS.h>`Inheritance diagram for CCfits::FITS::CantCreate::



### Public Member Functions

- [CantCreate](#) (const String &diag, bool silent=false)

*Exception ctor prefixes the string: "FITS Error: Cannot create file " before specific message.*

### 24.5.1 Detailed Description

thrown on failure to create new file

### 24.5.2 Constructor & Destructor Documentation

#### 24.5.2.1 CCfits::FITS::CantCreate::CantCreate (const String & msg, bool silent = false)

Exception ctor prefixes the string: "FITS Error: Cannot create file " before specific message. This exception will be thrown if the user attempts to write to a protected

directory or attempts to create a new file with the same name as an existing file without specifying overwrite [overwrite is specified by adding the character '!' before the filename, following the cfitsio convention].

#### Parameters:

- msg* A specific diagnostic message, the name of the file that was to be created.
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

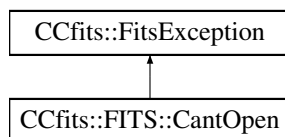
The documentation for this class was generated from the following files:

- FITS.h
- FITS.cxx

## 24.6 CCfits::FITS::CantOpen Class Reference

thrown on failure to open existing file

#include <FITS.h> Inheritance diagram for CCfits::FITS::CantOpen::



### Public Member Functions

- [CantOpen](#) (const String &diag, bool silent=true)  
*Exception ctor prefixes the string: "FITS Error: Cannot create file " before specific message.*

#### 24.6.1 Detailed Description

thrown on failure to open existing file

#### 24.6.2 Constructor & Destructor Documentation

##### 24.6.2.1 CCfits::FITS::CantOpen::CantOpen (const String &diag, bool silent = true)

Exception ctor prefixes the string: "FITS Error: Cannot create file " before specific message. This exception will be thrown if users attempt to open an existing file for write access to which they do not have permission, or of course if the file does not exist.

**Parameters:**

- diag* A specific diagnostic message, the name of the file that was to be created.
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- FITS.h
- FITS.cxx

## 24.7 CCfits::Column Class Reference

Abstract base class for [Column](#) objects.

```
#include <Column.h>
```

Inherited by [CCfits::ColumnData< T >](#), and [CCfits::ColumnVectorData< T >](#).

**Classes**

- class [InsufficientElements](#)  
*Exception thrown if the data supplied for a write operation is less than declared.*
- class [InvalidDataType](#)  
*Exception thrown for invalid data type inputs.*
- class [InvalidNumberOfRows](#)  
*Exception thrown if user enters a non-positive number for the number of rows to write.*
- class [InvalidRowNumber](#)  
*Exception thrown on attempting to read a row number beyond the end of a table.*
- class [InvalidRowParameter](#)  
*Exception thrown on incorrect row writing request.*
- class [NoNullValue](#)  
*Exception thrown if a null value is specified without support from existing column header.*

- class [RangeError](#)  
*exception to be thrown for inputs that cause range errors in column read operations.*
- class [WrongColumnType](#)  
*Exception thrown on attempting to access a scalar column as vector data.*

### Public Member Functions

- [Column](#) (const [Column](#) &right)  
*copy constructor, used in copying Columns to standard library containers.*
- virtual [~Column](#) ()  
*destructor.*
- template<typename T >  
void [addNullValue](#) (T nullVal)  
*Set the TNULLn keyword for the column.*
- const String & [dimen](#) () const  
*return TDIMn keyword*
- const String & [display](#) () const  
*return TDISPn keyword*
- const String & [format](#) () const  
*return TFORMn keyword*
- template<typename T >  
bool [getNullValue](#) (T \*nullVal) const  
*Get the value of the TNULLn keyword for the column.*
- int [index](#) () const  
*get the [Column](#) index (the n in TTYPEEn etc).*
- bool [isRead](#) () const  
*flag set to true if the entire column data has been read from disk*
- const String & [name](#) () const  
*return name of [Column](#) (TTYPEEn keyword)*
- [Table](#) \* [parent](#) () const

return a pointer to the [Table](#) which owns this [Column](#)

- template<typename S >  
void [read](#) (std::valarray< S > &vals, long rows, S \*nullValue)  
*return a single row of a vector column into a std::valarray, setting undefined values*
- template<typename S >  
void [read](#) (std::valarray< S > &vals, long first, long last, S \*nullValue)  
*Retrieve data from a scalar column into a std::valarray, applying nullValue when relevant.*
- template<typename S >  
void [read](#) (std::vector< S > &vals, long first, long last, S \*nullValue)  
*Retrieve data from a scalar column into a std::vector, applying nullValue when relevant.*
- template<typename S >  
void [read](#) (std::valarray< S > &vals, long rows)  
*return a single row of a vector column into a std::valarray*
- template<typename S >  
void [read](#) (std::valarray< S > &vals, long first, long last)  
*Retrieve data from a scalar column into a std::valarray.*
- template<typename S >  
void [read](#) (std::vector< S > &vals, long first, long last)  
*Retrieve data from a scalar column into a std::vector.*
- template<typename S >  
void [readArrays](#) (std::vector< std::valarray< S > > &vals, long first, long last, S \*nullValue)  
*return a set of rows of a vector column into a container, setting undefined values*
- template<typename S >  
void [readArrays](#) (std::vector< std::valarray< S > > &vals, long first, long last)  
*return a set of rows of a vector column into a vector of valarrays*
- virtual void [readData](#) (long firstRow, long nelements, long firstElem=1)=0  
*Read (or reread) data from the disk into the [Column](#) object's internal arrays.*
- size\_t [repeat](#) () const  
*get the repeat count for the rows*
- void [resetRead](#) ()

*reset the Column's isRead flag to false*

- int `rows ()` const  
*return the number of rows in the table.*
- double `scale ()` const  
*get TSCALn value*
- virtual void `setDimen ()`  
*set the TDIMn keyword.*
- void `setDisplay ()`  
*set the TDISPn keyword*
- ValueType `type ()` const  
*returns the data type of the column*
- const String & `unit ()` const  
*get units of data in Column (TUNITn keyword)*
- bool `varLength ()` const  
*boolean, set to true if Column has variable length vector rows.*
- long `width ()` const  
*return column data width*
- template<typename S >  
void `write` (S \*indata, long nElements, const std::vector< long > &vectorLengths, long firstRow)  
*write a C-array of values of size nElements into a vector column with specified number of entries written per row.*
- template<typename S >  
void `write` (const std::vector< S > &indata, const std::vector< long > &vectorLengths, long firstRow)  
*write a vector of values into a column with specified number of entries written per row.*
- template<typename S >  
void `write` (const std::valarray< S > &indata, const std::vector< long > &vectorLengths, long firstRow)  
*write a valarray of values into a column with specified number of entries written per row.*

- template<typename S >  
void [write](#) (S \*indata, long nElements, long nRows, long firstRow, S \*nullValue)  
*write a C array of values into a range of rows of a vector column, processing undefined values.*
- template<typename S >  
void [write](#) (const std::vector< S > &indata, long nRows, long firstRow, S \*nullValue)  
*write a vector of values into a range of rows of a vector column, processing undefined values*
- template<typename S >  
void [write](#) (const std::valarray< S > &indata, long nRows, long firstRow, S \*nullValue)  
*write a valarray of values into a range of rows of a vector column.*
- template<typename S >  
void [write](#) (S \*indata, long nElements, long nRows, long firstRow)  
*write a C array of values into a range of rows of a vector column*
- template<typename S >  
void [write](#) (const std::vector< S > &indata, long nRows, long firstRow)  
*write a vector of values into a range of rows of a vector column*
- template<typename S >  
void [write](#) (const std::valarray< S > &indata, long nRows, long firstRow)  
*write a valarray of values into a range of rows of a vector column.*
- template<typename S >  
void [write](#) (S \*indata, long nRows, long firstRow, S \*nullValue)  
*write a C array into a scalar [Column](#), processing undefined values.*
- template<typename S >  
void [write](#) (const std::valarray< S > &indata, long firstRow, S \*nullValue)  
*write a valarray of values into a scalar column starting with firstRow, replacing elements equal to nullValue with the [FITS](#) null value.*
- template<typename S >  
void [write](#) (const std::vector< S > &indata, long firstRow, S \*nullValue)  
*write a vector of values into a scalar column starting with firstRow, replacing elements equal to nullValue with the [FITS](#) null value.*



- template<typename S >  
void [write](#) (S \*indata, long nRows, long firstRow)  
*write a C array of size nRows into a scalar [Column](#) starting with row firstRow.*
- template<typename S >  
void [write](#) (const std::valarray< S > &indata, long firstRow)  
*write a valarray of values into a scalar column starting with firstRow*
- template<typename S >  
void [write](#) (const std::vector< S > &indata, long firstRow)  
*write a vector of values into a scalar column starting with firstRow*
- template<typename S >  
void [writeArrays](#) (const std::vector< std::valarray< S > > &indata, long firstRow, S \*nullValue)  
*write a vector of valarray objects to the column, starting at row firstRow >= 1, processing undefined values*
- template<typename S >  
void [writeArrays](#) (const std::vector< std::valarray< S > > &indata, long firstRow)  
*write a vector of valarray objects to the column, starting at row firstRow >= 1*
- double [zero](#) () const  
*get TZERO value*

### Protected Member Functions

- [Column](#) ([Table](#) \*p=0)  
*Simple constructor to be called by subclass reading ctors.*
- [Column](#) (int columnIndex, const String &columnName, ValueType type, const String &format, const String &unit, [Table](#) \*p, int rpt=1, long w=1, const String &comment="")  
*new column creation constructor*
- const String & [comment](#) () const  
*retrieve comment for [Column](#)*
- fitsfile \* [fitsPointer](#) ()  
*fits pointer corresponding to fits file containing column data.*

- void [makeHDUCurrent](#) ()  
*make [HDU](#) containing this the current [HDU](#) of the fits file.*
- virtual std::ostream & [put](#) (std::ostream &s) const  
*internal implementation of << operator.*

### 24.7.1 Detailed Description

Abstract base class for [Column](#) objects. Columns are the data containers used in [FITS](#) tables. Columns of scalar type (one entry per cell) are implemented by the template subclass `ColumnData<T>`. Columns of vector type (vector and variable rows) are implemented with the template subclass `ColumnVectorData<T>`. `AsciiTables` may only contain Columns of type `ColumnData<T>`, where T is an implemented [FITS](#) data type (see the `CCfits.h` header for a complete list. This requirement is enforced by ensuring that `AsciiTable`'s `addColumn` method may only create an [AsciiTable](#) compatible column. The `ColumnData<T>` class stores its data in a `std::vector<T>` object.

`BinTables` may contain either `ColumnData<T>` or `ColumnVectorData<T>`. For `ColumnVectorData`, T must be a numeric type: string vectors are handled by `ColumnData<T>`; string arrays are not supported. The internal representation of the data is a `std::vector<std::valarray<T> >` object. The `std::valarray` class is designed for efficient numeric processing and has many vectorized numeric and transcendental functions defined on it.

Member template functions for read/write operations are provided in multiple overloads as the interface to data operations. Implicit data type conversions are supported but where they are required make the operations less efficient. Reading numeric column data as character arrays, supported by `cfitsio`, is not supported by `CCfits`.

As a base class, [Column](#) provides protected accessor/mutator inline functions to allow only its subclasses to access data members.

### 24.7.2 Constructor & Destructor Documentation

#### 24.7.2.1 CCfits::Column::Column (const Column & right)

copy constructor, used in copying Columns to standard library containers. The copy constructor is for internal use only: it does not affect the disk fits file associated with the object.

**24.7.2.2 CCfits::Column::Column (int *columnIndex*, const String & *columnName*, ValueType *type*, const String & *format*, const String & *unit*, Table \* *p*, int *rpt* = 1, long *w* = 1, const String & *comment* = "")**  
**[protected]**

new column creation constructor This constructor allows the specification of:

**Parameters:**

*columnIndex* The column number  
*columnName* The column name, keyword TTYPEn  
*type* used for determining class of T in ColumnData<T>, ColumnVectorData<T>  
*format* the column data format, TFORMn keyword  
*unit* the column data unit, TUNITn keyword  
*p* the Table pointer  
*rpt* (optional) repeat count for the row ( == 1 for AsciiTables)  
*w* the row width  
*comment* comment to be added to the header.

### 24.7.3 Member Function Documentation

**24.7.3.1 template<typename T > void CCfits::Column::addNullValue (T *nullVal*)** **[inline]**

Set the TNULLn keyword for the column. Only relevant for integer valued columns, TNULLn is the value used by cfitsio in undefined processing. All entries in the table equal to an input "null value" are set equal to the value of TNULLn. (For floating point columns a system NaN value is used).

**24.7.3.2 const String & CCfits::Column::dimen () const** **[inline]**

return TDIMn keyword represents dimensions of data arrays in vector columns. for scalar columns, returns a default value.

**24.7.3.3 const String & CCfits::Column::display () const [inline]**

return TDISPn keyword TDISPn is suggested format for output of column data.

**24.7.3.4 const String & CCfits::Column::format () const [inline]**

return TFORMn keyword TFORMn specifies data format stored in disk file.

**24.7.3.5 template<typename T > bool CCfits::Column::getNullValue (T \* nullVal) const [inline]**

Get the value of the TNULLn keyword for the column. Only relevant for integer valued columns. If the TNULLn keyword is present, its value will be written to *\*nullVal* and the function returns *true*. If the keyword is not found or its value is undefined, the function returns *false* and *\*nullVal* is not modified.

**Parameters:**

*nullVal* A pointer to the variable for storing the TNULLn value.

**24.7.3.6 template<typename S > void CCfits::Column::read (std::valarray< S > & vals, long first, long last, S \* nullValue) [inline]**

Retrieve data from a scalar column into a std::valarray, applying nullValue when relevant. If both *nullValue* and *\*nullValue* are not 0, then any undefined values in the file will be converted to *\*nullValue* when copied into the *vals* valarray. See cfitsio documentation for further details

**Parameters:**

*vals* The output container. The function will resize this as necessary

*first,last* the span of row numbers to read.

*nullValue* pointer to value to be applied to undefined elements.

**24.7.3.7** `template<typename S> void CCfits::Column::read (std::vector< S > & vals, long first, long last, S * nullValue) [inline]`

Retrieve data from a scalar column into a `std::vector<`, applying `nullValue` when relevant. If both `nullValue` and `*nullValue` are not 0, then any undefined values in the file will be converted to `*nullValue` when copied into the `vals` vector. See `cfitsio` documentation for further details

**Parameters:**

*vals* The output container. The function will resize this as necessary

*first,last* the span of row numbers to read.

*nullValue* pointer to value to be applied to undefined elements.

**24.7.3.8** `template<typename S> void CCfits::Column::read (std::valarray< S > & vals, long rows) [inline]`

return a single row of a vector column into a `std::valarray`

**Parameters:**

*vals* The output valarray object

*rows* The row number to be retrieved (starting at 1).

**24.7.3.9** `template<typename S> void CCfits::Column::read (std::valarray< S > & vals, long first, long last) [inline]`

Retrieve data from a scalar column into a `std::valarray`.

**Parameters:**

*vals* The output container. The function will resize this as necessary

*first,last* the span of row numbers to read.

**24.7.3.10** `template<typename S> void CCfits::Column::read (std::vector< S > & vals, long first, long last) [inline]`

Retrieve data from a scalar column into a `std::vector`. This and the following functions perform implicit data conversions. An exception will be thrown if no conversion exists.

**Parameters:**

*vals* The output container. The function will resize this as necessary  
*first,last* the span of row numbers to read.

**24.7.3.11** `template<typename S > void CCfits::Column::readArrays  
 (std::vector< std::valarray< S > > & vals, long first, long last, S *  
 nullValue) [inline]`

return a set of rows of a vector column into a container, setting undefined values

**Parameters:**

*vals* The output container. The function will resize this as necessary  
*first,last* the span of row numbers to read.  
*nullValue* pointer to integer value regarded as undefined

**24.7.3.12** `template<typename S > void CCfits::Column::readArrays  
 (std::vector< std::valarray< S > > & vals, long first, long last)  
 [inline]`

return a set of rows of a vector column into a vector of valarrays

**Parameters:**

*vals* The output container. The function will resize this as necessary  
*first,last* the span of row numbers to read.

**24.7.3.13** `void CCfits::Column::readData (long firstRow = 1, long nelements =  
 1, long firstElem = 1) [pure virtual]`

Read (or reread) data from the disk into the [Column](#) object's internal arrays. This function normally does not need to be called. See the **resetRead** function for an alternative way of performing a reread from disk.

**Parameters:**

*firstRow* The first row to be read

*nElements* The number of elements to read

*firstElem* The number of the element on the first row to start at (ignored for scalar columns)

**24.7.3.14 void CCfits::Column::resetRead () [inline]**

reset the Column's isRead flag to false This forces the data to be reread from the disk the next time a read command is called on the [Column](#), rather than simply retrieving the data already stored in the [Column](#) object's internal arrays. This may be useful for example if trying to reread a [Column](#) using a different nullValue argument than for an earlier read.

**24.7.3.15 int CCfits::Column::rows () const**

return the number of rows in the table. return number of rows in the [Column](#)

**24.7.3.16 double CCfits::Column::scale () const [inline]**

get TSCALn value TSCALn is used to convert a data array represented on disk in integer format as floating. Useful for compact storage of digitized data.

**24.7.3.17 template<typename S > void CCfits::Column::write (S \* indata, long nElements, const std::vector< long > & vectorLengths, long firstRow) [inline]**

write a C-array of values of size nElements into a vector column with specified number of entries written per row. Intended for writing a varying number of elements to multiple rows in a vector column, this may be used for either variable or fixed-width columns. See the indata valarray version of this function for a complete description.

**24.7.3.18** `template<typename S > void CCfits::Column::write (const  
std::vector< S > & indata, const std::vector< long > &  
vectorLengths, long firstRow) [inline]`

write a vector of values into a column with specified number of entries written per row. Intended for writing a varying number of elements to multiple rows in a vector column, this may be used for either variable or fixed-width columns. See the *indata* valarray version of this function for a complete description.

**24.7.3.19** `template<typename S > void CCfits::Column::write (const  
std::valarray< S > & indata, const std::vector< long > &  
vectorLengths, long firstRow) [inline]`

write a valarray of values into a column with specified number of entries written per row. Data are written into *vectorLengths.size()* rows, with *vectorLength[n]* elements written to row *n+firstRow -1*. Although primarily intended for wrapping calls to multiple variable-width vector column rows, it may also be used to write a variable number of elements to fixed-width column rows.

When writing to fixed-width column rows, if the number of elements sent to a particular row are fewer than the column's repeat value, the remaining elements in the row will not be modified.

Since *ccfitsio* does not support null value processing for variable width columns this function and its variants do not have version which process undefined values

#### Parameters:

*indata* The data to be written

*vectorLengths* the number of elements to write to each successive row.

*firstRow* the first row to be written.

**24.7.3.20** `template<typename S > void CCfits::Column::write (S * indata, long  
nElements, long nRows, long firstRow, S * nullValue) [inline]`

write a C array of values into a range of rows of a vector column, processing undefined values. see version without undefined processing for details.



**24.7.3.21** `template<typename S > void CCfits::Column::write (const  
std::vector< S > & indata, long nRows, long firstRow, S * nullValue)  
[inline]`

write a vector of values into a range of rows of a vector column, processing undefined values see version without undefined processing for details.

**24.7.3.22** `template<typename S > void CCfits::Column::write (const  
std::valarray< S > & indata, long nRows, long firstRow, S *  
nullValue) [inline]`

write a valarray of values into a range of rows of a vector column. see version without undefined processing for details.

**24.7.3.23** `template<typename S > void CCfits::Column::write (S * indata,  
long nElements, long nRows, long firstRow) [inline]`

write a C array of values into a range of rows of a vector column Details are as for vector input; only difference is the need to supply the size of the C-array.

#### Parameters:

*indata* The data to be written.

*nElements* The size of *indata*

*nRows* the number of rows to which to write the data.

*firstRow* The first row to be written

**24.7.3.24** `template<typename S > void CCfits::Column::write (const  
std::vector< S > & indata, long nRows, long firstRow) [inline]`

write a vector of values into a range of rows of a vector column The primary use of this is for fixed width columns, in which case Column's repeat attribute is used to determine how many elements are written to each row; if *indata.size()* is too small an exception will be thrown. If the column is variable width, the call will write *indata.size()/nRows* elements to each row.

**Parameters:**

*indata* The data to be written.  
*nRows* the number of rows to which to write the data.  
*firstRow* The first row to be written

**24.7.3.25** `template<typename S > void CCfits::Column::write (const  
 std::valarray< S > & indata, long nRows, long firstRow)  
 [inline]`

write a valarray of values into a range of rows of a vector column. The primary use of this is for fixed width columns, in which case Column's repeat attribute is used to determine how many elements are written to each row; if indata.size() is too small an exception will be thrown. If the column is variable width, the call will write indata.size()/nRows elements to each row.

**Parameters:**

*indata* The data to be written.  
*nRows* the number of rows to which to write the data.  
*firstRow* The first row to be written

**24.7.3.26** `template<typename S > void CCfits::Column::write (S * indata,  
 long nRows, long firstRow, S * nullValue) [inline]`

write a C array into a scalar [Column](#), processing undefined values.

**Parameters:**

*indata* The data to be written.  
*nRows* The size of the data array to be written  
*firstRow* The first row to be written  
*nullValue* Pointer to the value in the input array to be set to undefined values

**24.7.3.27** `template<typename S > void CCfits::Column::write (const  
 std::valarray< S > & indata, long firstRow, S * nullValue)  
 [inline]`

write a valarray of values into a scalar column starting with *firstRow*, replacing elements equal to *nullValue* with the [FITS](#) null value. If *nullValue* is not 0, the appropriate [FITS](#) null value will be substituted for all elements of *indata* equal to *\*nullValue*. For integer type columns there must be a pre-existing TNULLn keyword to define the [FITS](#) null value, otherwise a [FitsError](#) exception is thrown. For floating point columns, the [FITS](#) null is the IEEE NaN (Not-a-Number) value. See the cfitsio fits\_write\_colnull function documentation for more details.

**Parameters:**

*indata* The data to be written.

*firstRow* The first row to be written

*nullValue* Pointer to the value for which equivalent *indata* elements will be replaced in the file with the appropriate [FITS](#) null value.

**24.7.3.28** `template<typename S > void CCfits::Column::write (const std::vector< S > & indata, long firstRow, S * nullValue) [inline]`

write a vector of values into a scalar column starting with *firstRow*, replacing elements equal to *nullValue* with the [FITS](#) null value. If *nullValue* is not 0, the appropriate [FITS](#) null value will be substituted for all elements of *indata* equal to *\*nullValue*. For integer type columns there must be a pre-existing TNULLn keyword to define the [FITS](#) null value, otherwise a [FitsError](#) exception is thrown. For floating point columns, the [FITS](#) null is the IEEE NaN (Not-a-Number) value. See the cfitsio fits\_write\_colnull function documentation for more details.

**Parameters:**

*indata* The data to be written.

*firstRow* The first row to be written

*nullValue* Pointer to the value for which equivalent *indata* elements will be replaced in the file with the appropriate [FITS](#) null value.

**24.7.3.29** `template<typename S > void CCfits::Column::write (S * indata, long nRows, long firstRow) [inline]`

write a C array of size *nRows* into a scalar [Column](#) starting with row *firstRow*.

**Parameters:**

*indata* The data to be written.

*nRows* The size of the data array to be written

*firstRow* The first row to be written

**24.7.3.30** `template<typename S > void CCfits::Column::write (const  
std::valarray< S > & indata, long firstRow) [inline]`

write a valarray of values into a scalar column starting with firstRow

**Parameters:**

*indata* The data to be written.

*firstRow* The first row to be written

**24.7.3.31** `template<typename S > void CCfits::Column::write (const  
std::vector< S > & indata, long firstRow) [inline]`

write a vector of values into a scalar column starting with firstRow

**Parameters:**

*indata* The data to be written.

*firstRow* The first row to be written

**24.7.3.32** `template<typename S > void CCfits::Column::writeArrays (const  
std::vector< std::valarray< S > > & indata, long firstRow, S *  
nullValue) [inline]`

write a vector of valarray objects to the column, starting at row firstRow >= 1, processing undefined values see version without undefined processing for details.

**24.7.3.33** `template<typename S > void CCfits::Column::writeArrays (const  
std::vector< std::valarray< S > > & indata, long firstRow)  
[inline]`

write a vector of valarray objects to the column, starting at row firstRow >= 1 Intended for writing a varying number of elements to multiple rows in a vector column, this

may be used for either variable or fixed-width columns. When writing to fixed-width column rows, if the number of elements sent to a particular row are fewer than the column's repeat value, the remaining elements in the row will not be modified.

#### Parameters:

- indata* The data to be written
- firstRow* the first row to be written.

#### 24.7.3.34 double CCfits::Column::zero () const [inline]

get TZEROn value TZEROn is an integer offset used in the implementation of unsigned data

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx
- ColumnT.h

## 24.8 CCfits::FITSUtil::CVAarray< T > Class Template Reference

function object returning C array from a vector of valarrays. see [CVarray](#) for details

#include <FITSUtil.h>

### Public Member Functions

- [T \\* operator\(\)](#) (const std::vector< std::valarray< T > > &inArray)  
*operator returning C array for use with vector column data.*

#### 24.8.1 Detailed Description

**template<typename T> class CCfits::FITSUtil::CVAarray< T >**

function object returning C array from a vector of valarrays. see [CVarray](#) for details

The documentation for this class was generated from the following file:

- FITSUtil.h

## 24.9 CCfits::FITSUtil::CArray< T > Class Template Reference

Function object class for returning C arrays from standard library objects used in the [FITS](#) library implementation.

```
#include <FITSUtil.h>
```

### Public Member Functions

- `T * operator()` (const std::vector< T > &inArray)  
*operator returning C array for use with scalar column data.*

### 24.9.1 Detailed Description

```
template<typename T> class CCfits::FITSUtil::CArray< T >
```

Function object class for returning C arrays from standard library objects used in the [FITS](#) library implementation. There are 3 versions which convert std::vector<T>, std::valarray<T>, and std::vector<std::valarray<T> > objects to pointers to T, called [CArray](#), [CAarray](#), and [CVAarray](#).

An alternative function, CharArray, is provided to deal with the special case of vector string arrays.

The documentation for this class was generated from the following file:

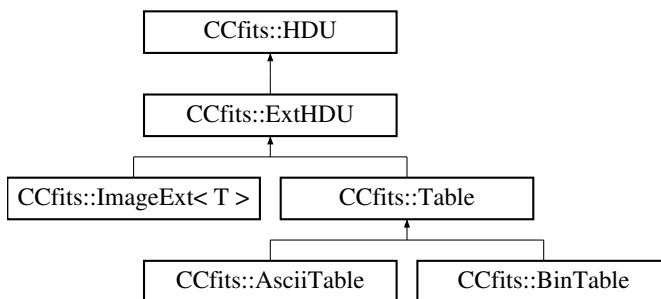
- FITSUtil.h

## 24.10 CCfits::ExtHDU Class Reference

base class for all [FITS](#) extension HDUs, i.e. Image Extensions and Tables.

```
#include <ExtHDU.h>
```

Inheritance diagram for CCfits::ExtHDU::



## Classes

- class [WrongExtensionType](#)  
*Exception to be thrown on unmatched extension types.*

## Public Member Functions

- [ExtHDU](#) (const [ExtHDU](#) &right)  
*copy constructor*
- virtual [~ExtHDU](#) ()  
*destructor*
- virtual void [addColumn](#) (ValueType type, const String &columnName, long repeatWidth, const String &colUnit=String(""), long decimals=-1, size\_t columnNumber=0)  
*add a new column to an existing table [HDU](#).*
- virtual [HDU](#) \* [clone](#) (FITSBase \*p) const =0  
*virtual copy constructor*
- virtual const std::map< string, [Column](#) \* > & [column](#) () const  
*return a reference to the array containing the columns.*
- virtual [Column](#) & [column](#) (int colIndex) const  
*return a reference to a [Table](#) column specified by column index.*
- virtual [Column](#) & [column](#) (const String &colName, bool caseSensitive=true) const  
*return a reference to a [Table](#) column specified by name.*
- virtual void [deleteColumn](#) (const String &columnName)  
*delete a column in a [Table](#) extension by name.*
- virtual long [getRowsize](#) () const  
*return the optimal number of rows to read or write at a time*
- virtual void [makeThisCurrent](#) () const  
*move the fitsfile pointer to this current [HDU](#).*
- const String & [name](#) () const  
*return the name of the extension.*

- virtual int [numCols](#) () const  
*return the number of Columns in the [Table](#) (the `TFIELDS` keyword).*
- template<typename S >  
void [read](#) (std::valarray< S > &image, const std::vector< long > &firstVertex, const std::vector< long > &lastVertex, const std::vector< long > &stride, S \*nullValue)  
*read an image subset into valarray image, processing null values*
- template<typename S >  
void [read](#) (std::valarray< S > &image, const std::vector< long > &first, long nElements)  
*read an image section starting at a location specified by an n-tuple*
- template<typename S >  
void [read](#) (std::valarray< S > &image, long first, long nElements)  
*read an image section starting at a specified pixel*
- template<typename S >  
void [read](#) (std::valarray< S > &image, const std::vector< long > &firstVertex, const std::vector< long > &lastVertex, const std::vector< long > &stride)  
*read an image subset*
- template<typename S >  
void [read](#) (std::valarray< S > &image, const std::vector< long > &first, long nElements, S \*nullValue)  
*read part of an image array, processing null values.*
- template<typename S >  
void [read](#) (std::valarray< S > &image, long first, long nElements, S \*nullValue)  
*read part of an image array, processing null values.*
- template<typename S >  
void [read](#) (std::valarray< S > &image)  
*Read image data into container.*
- virtual void [readData](#) (bool readFlag=false, const std::vector< String > &keys=std::vector< String >())=0  
*read data from [HDU](#) depending on readFlag and keys.*
- virtual long [rows](#) () const



*return the number of rows in the extension.*

- void **version** (int value)  
*set the extension version number*
- int **version** () const  
*return the extension version number.*
- template<typename S >  
void **write** (const std::vector< long > &firstVertex, const std::vector< long > &lastVertex, const std::valarray< S > &data)  
*write a subset (generalize slice) of data to the image*
- template<typename S >  
void **write** (long first, long nElements, const std::valarray< S > &data)  
*write array starting from specified pixel number, without undefined data processing*
- template<typename S >  
void **write** (const std::vector< long > &first, long nElements, const std::valarray< S > &data)  
*write array starting from specified n-tuple, without undefined data processing*
- template<typename S >  
void **write** (long first, long nElements, const std::valarray< S > &data, S \*nullValue)  
*write array to image starting with a specified pixel and allowing undefined data to be processed*
- template<typename S >  
void **write** (const std::vector< long > &first, long nElements, const std::valarray< S > &data, S \*nullValue)  
*Write a set of pixels to an image extension with the first pixel specified by an n-tuple, processing undefined data.*

### Static Public Member Functions

- static void **readHduName** (const fitsfile \*fptr, int hduIndex, String &hduName, int &hduVersion)  
*read extension name.*

### Protected Member Functions

- [ExtHDU](#) (FITSBase \*p, HduType xtype, int number)  
*ExtHDU constructor for getting ExtHDUs by number.*
- [ExtHDU](#) (FITSBase \*p, HduType xtype, const String &hduName, int bitpix, int naxis, const std::vector< long > &axes, int version)  
*writing constructor.*
- [ExtHDU](#) (FITSBase \*p, HduType xtype, const String &hduName, int version)  
*default constructor, required as Standard Library Container content.*
- void [gcount](#) (long value)  
*set required gcount keyword value*
- long [gcount](#) () const  
*return required gcount keyword value*
- void [pcount](#) (long value)  
*set required pcount keyword value*
- long [pcount](#) () const  
*return required pcount keyword value*
- void [xtension](#) (HduType value)  
*set the extension type*
- HduType [xtension](#) () const  
*return the extension type*

#### 24.10.1 Detailed Description

base class for all [FITS](#) extension HDUs, i.e. Image Extensions and Tables. [ExtHDU](#) needs to have the combined public interface of [Table](#) objects and images. It achieves this by providing the same set of read and write operations as [PHDU](#), and also providing the same operations for extracting columns from the extension as does [Table](#) [after which the column interface is accessible]. Differentiation between extension types operates by exception handling: .i.e. attempting to access image data structures on a [Table](#) object through the [ExtHDU](#) interface will or trying to return a [Column](#) reference from an Image extension will both throw an exception

### 24.10.2 Constructor & Destructor Documentation

**24.10.2.1** `CCfits::ExtHDU::ExtHDU (FITSBase * p, HduType xtype, const String & hduName, int bitpix, int naxis, const std::vector< long > & axes, int version)` `[protected]`

writing constructor. The writing constructor forces the user to supply a name for the [HDU](#). The *bitpix*, *naxes* and *naxis* data required by this constructor are required [FITS](#) keywords for any HDUs.

**24.10.2.2** `CCfits::ExtHDU::ExtHDU (FITSBase * p, HduType xtype, int number)` `[protected]`

[ExtHDU](#) constructor for getting ExtHDUs by number. Necessary since EXTNAME is a reserved, not required, keyword. But a synthetic name is supplied by static [ExtHDU::readHduName](#) which is called by this constructor.

### 24.10.3 Member Function Documentation

**24.10.3.1** `void CCfits::ExtHDU::addColumn (ValueType type, const String & columnName, long repeatWidth, const String & colUnit = String(""), long decimals = -1, size_t columnNumber = 0)` `[virtual]`

add a new column to an existing table [HDU](#).

#### Parameters:

*type* The data type of the column to be added

*columnName* The name of the column to be added

*repeatWidth* for a string valued, this is the width of a string. For a numeric column it supplies the vector length of the rows. It is ignored for ascii table numeric data.

*colUnit* an optional field specifying the units of the data (TUNITn)

*decimals* optional parameter specifying the number of decimals for an ascii numeric column

*columnNumber* optional parameter specifying column number to be created. If not specified the column is added to the end. If specified, the column is

inserted and the columns already read are reindexed. This parameter is provided as a convenience to support existing code rather than recommended.

Reimplemented in [CCfits::AsciiTable](#), and [CCfits::BinTable](#).

#### 24.10.3.2 `const map< string, Column * > & CCfits::ExtHDU::column () const [virtual]`

return a reference to the array containing the columns.

##### Exceptions:

[\*WrongExtensionType\*](#) thrown if \*this is an image extension.

Reimplemented in [CCfits::Table](#).

#### 24.10.3.3 `Column & CCfits::ExtHDU::column (int colIndex) const [virtual]`

return a reference to a [Table](#) column specified by column index. This version is provided for convenience; the 'return by name' version is more efficient because columns are stored in an associative array sorted by name.

##### Exceptions:

[\*WrongExtensionType\*](#) thrown if \*this is an image extension.

Reimplemented in [CCfits::Table](#).

#### 24.10.3.4 `Column & CCfits::ExtHDU::column (const String & colName, bool caseSensitive = true) const [virtual]`

return a reference to a [Table](#) column specified by name. If the *caseSensitive* parameter is set to false, the search will be case-insensitive. The overridden base class implementation [ExtHDU::column](#) throws an exception, which is thus the action to be taken if self is an image extension

##### Exceptions:

[\*WrongExtensionType\*](#) see above

Reimplemented in [CCfits::Table](#).

### 24.10.3.5 void CCfits::ExtHDU::deleteColumn (const String & *columnName*) [virtual]

delete a column in a [Table](#) extension by name.

#### Parameters:

*columnName* The name of the column to be deleted.

#### Exceptions:

[WrongExtensionType](#) if extension is an image.

Reimplemented in [CCfits::Table](#).

### 24.10.3.6 long CCfits::ExtHDU::getRowsize () const [virtual]

return the optimal number of rows to read or write at a time A wrapper for the CFIT-SIO function fits\_get\_rowsize, useful for obtaining maximum I/O efficiency. This will throw if it is not called for a [Table](#) extension.

Reimplemented in [CCfits::Table](#).

### 24.10.3.7 void CCfits::ExtHDU::makeThisCurrent () const [virtual]

move the fitsfile pointer to this current [HDU](#). This function should never need to be called by the user since it is called internally whenever required.

Reimplemented from [CCfits::HDU](#).

### 24.10.3.8 int CCfits::ExtHDU::numCols () const [virtual]

return the number of Columns in the [Table](#) (the TFIELDSD keyword).

#### Exceptions:

[WrongExtensionType](#) thrown if \*this is an image extension.

Reimplemented in [CCfits::Table](#).

**24.10.3.9** `template<typename S> void CCfits::ExtHDU::read (std::valarray< S> & image, const std::vector< long> & firstVertex, const std::vector< long> & lastVertex, const std::vector< long> & stride, S * nullValue) [inline]`

read an image subset into valarray image, processing null values. The image subset is defined by two vertices and a stride indicating the 'denseness' of the values to be picked in each dimension (a stride = (1,1,1,...) means picking every pixel in every dimension, whereas stride = (2,2,2,...) means picking every other value in each dimension.

**24.10.3.10** `template<typename S> void CCfits::ExtHDU::read (std::valarray< S> & image, const std::vector< long> & first, long nElements, S * nullValue) [inline]`

read part of an image array, processing null values. As above except for

**Parameters:**

*first* a vector<long> representing an n-tuple giving the coordinates in the image of the first pixel.

**24.10.3.11** `template<typename S> void CCfits::ExtHDU::read (std::valarray< S> & image, long first, long nElements, S * nullValue) [inline]`

read part of an image array, processing null values. Implicit data conversion is supported (i.e. user does not need to know the type of the data stored. A [WrongExtension-Type](#) extension is thrown if \*this is not an image.

**Parameters:**

*image* The receiving container, a std::valarray reference

*first* The first pixel from the array to read [a long value]

*nElements* The number of values to read

*nullValue* A pointer containing the value in the table to be considered as undefined. See cfitsio for details

**24.10.3.12** `template<typename S> void CCfits::ExtHDU::read  
(std::valarray< S> & image) [inline]`

Read image data into container. The container image contains the entire image array after the call. This and all the other variants of `read()` throw a [WrongExtensionType](#) exception if called for a [Table](#) object.

**24.10.3.13** `static void CCfits::ExtHDU::readHduName (const fitsfile * fptr, int  
hduIndex, String & hduName, int & hduVersion) [static]`

read extension name. Used primarily to allow extensions to be specified by [HDU](#) number and provide their name for the associative array that contains them. Alternatively, if there is no name keyword in the extension, one is synthesized from the index.

**24.10.3.14** `long CCfits::ExtHDU::rows () const [virtual]`

return the number of rows in the extension.

#### Exceptions:

[WrongExtensionType](#) thrown if \*this is an image extension.

Reimplemented in [CCfits::Table](#).

**24.10.3.15** `template<typename S> void CCfits::ExtHDU::write (const  
std::vector< long> & firstVertex, const std::vector< long> &  
lastVertex, const std::valarray< S> & data) [inline]`

write a subset (generalize slice) of data to the image A generalized slice/subset is a subset of the image (e.g. one plane of a data cube of size <= the dimension of the cube). It is specified by two opposite vertices. The equivalent cfitsio call does not support undefined data processing so there is no version that allows a null value to be specified.

#### Parameters:

*firstVertex* the coordinates specifying lower and upper vertices of the n-dimensional slice

*lastVertex*

*data* The data to be written

**24.10.3.16** `template<typename S > void CCfits::ExtHDU::write (long first,  
long nElements, const std::valarray< S > & data, S * nullValue)  
[inline]`

write array to image starting with a specified pixel and allowing undefined data to be processed parameters after the first are as for version with n-tuple specifying first element. these two version are equivalent, except that it is possible for the first pixel number to exceed the range of 32-bit integers, which is how long datatype is commonly implemented.

**24.10.3.17** `template<typename S > void CCfits::ExtHDU::write (const  
std::vector< long > & first, long nElements, const std::valarray< S  
> & data, S * nullValue) [inline]`

Write a set of pixels to an image extension with the first pixel specified by an n-tuple, processing undefined data. All the overloaded versions of [ExtHDU::write](#) perform operations on \*this if it is an image and throw a [WrongExtensionType](#) exception if not. Where appropriate, alternate versions allow undefined data to be processed

#### Parameters:

*first* an n-tuple of dimension equal to the image dimension specifying the first pixel in the range to be written

*nElements* number of pixels to be written

*data* array of data to be written

*nullValue* pointer to null value (data with this value written as undefined; needs the BLANK keyword to have been specified).

**24.10.3.18** `HduType CCfits::ExtHDU::xtension () const [inline,  
protected]`

return the extension type allowed values are ImageHDU, AsciiTbl, and BinaryTbl

The documentation for this class was generated from the following files:



- ExtHDU.h
- ExtHDU.cxx
- ExtHDUT.h

## 24.11 CCfits::FITS Class Reference

Memory object representation of a disk [FITS](#) file.

```
#include <FITS.h>
```

### Classes

- class [CantCreate](#)  
*thrown on failure to create new file*
- class [CantOpen](#)  
*thrown on failure to open existing file*
- class [NoSuchHDU](#)  
*exception thrown by [HDU](#) retrieval methods.*
- class [OperationNotSupported](#)  
*thrown for unsupported operations, such as attempted to select rows from an image extension.*

### Public Member Functions

- [FITS](#) (const String &name, RWmode mode, const std::vector< String > &searchKeys, const std::vector< String > &searchValues, bool readDataFlag=false, const std::vector< String > &hduKeys=std::vector< String >(), const std::vector< String > &primaryKey=std::vector< string >(), int version=1)  
*open fits file and read [HDU](#) which contains supplied keywords with [optional] specified values (sometimes one just wants to know that the keyword is present).*
- [FITS](#) (const string &name, RWmode mode, int hduIndex, bool readDataFlag=false, const std::vector< String > &hduKeys=std::vector< String >(), const std::vector< String > &primaryKey=std::vector< String >())  
*read a single numbered [HDU](#).*
- [FITS](#) (const String &name, int bitpix, int naxis, long \*naxes)  
*Constructor for creating new [FITS](#) objects containing images.*

- **FITS** (const String &name, RWmode mode, const std::vector< String > &hduNames, const std::vector< std::vector< String > > &hduKeys, bool readDataFlag=false, const std::vector< String > &primaryKeys=std::vector< String >(), const std::vector< int > &hduVersions=std::vector< int >())  
*FITS read constructor in full generality.*
- **FITS** (const String &fileName, const **FITS** &source)  
*create a new **FITS** object and corresponding file with copy of the primary header of the source. If the filename corresponds to an existing file and does not start with the '!' character the construction will fail with a **CantCreate** exception.*
- **FITS** (const String &name, RWmode mode, const std::vector< String > &hduNames, bool readDataFlag=false, const std::vector< String > &primaryKey=std::vector< String >())
- **FITS** (const String &name, RWmode mode, const string &hduName, bool readDataFlag=false, const std::vector< String > &hduKeys=std::vector< String >(), const std::vector< String > &primaryKey=std::vector< String >(), int version=1)  
*Open a **FITS** file and read a single specified **HDU**.*
- **FITS** (const String &name, RWmode mode=Read, bool readDataFlag=false, const std::vector< String > &primaryKeys=std::vector< String >())  
*basic constructor*
- **~FITS** ()  
*destructor*
- **ExtHDU** \* **addImage** (const String &hduName, int bpix, std::vector< long > &naxes, int version=1)  
*Add an image extension to an existing **FITS** object. (File with w or rw access).*
- **Table** \* **addTable** (const String &hduName, int rows, const std::vector< String > &columnName=std::vector< String >(), const std::vector< String > &columnFmt=std::vector< String >(), const std::vector< String > &columnUnit=std::vector< String >(), HduType type=BinaryTbl, int version=1)  
*Add a table extension to an existing **FITS** object. Add extension to **FITS** object for file with w or rw access.*
- void **copy** (const **HDU** &source)  
*copy the **HDU** source into the **FITS** object.*
- **ExtHDU** & **currentExtension** ()

*return a non-const reference to whichever is the current extension.*

- `const String & currentExtensionName () const`  
*return the name of the extension that the fitsfile is currently addressing.*
- `void deleteExtension (int doomed)`  
*Delete extension specified by extension number.*
- `void deleteExtension (const String &doomed, int version=1)`  
*Delete extension specified by name and version number.*
- `void destroy () throw ()`  
*Erase *FITS* object and close corresponding file.*
- `const ExtMap & extension () const`  
*return const reference to the extension container*
- `ExtHDU & extension (const String &hduName, int version=1)`  
*return *FITS* extension by name and (optionally) version number.*
- `const ExtHDU & extension (const String &hduName, int version=1) const`  
*return *FITS* extension by name and (optionally) version number.*
- `ExtHDU & extension (int i)`  
*return *FITS* extension by index number. non-const version. see const version for details.*
- `const ExtHDU & extension (int i) const`  
*return *FITS* extension by index number. N.B. The input index number is currently defined as enumerating extensions, so the extension(1) returns *HDU* number 2.*
- `Table & filter (const String &expression, ExtHDU &inputTable, bool overwrite=true, bool readData=false)`  
*Filter the rows of the inputTable with the condition expression, and return a reference to the resulting *Table*.*
- `fitsfile * fitsPointer () const`  
*return the CFITSIO fitsfile pointer for this *FITS* object*
- `void flush ()`  
*flush buffer contents to disk*
- `int getCompressionType () const`

*Get the int specifying the compression algorithm to be used when adding an image extension.*

- int [getNoiseBits](#) () const  
*Get the cfitsio noisebits parameter used when compressing floating-point images.*
- void [getTileDimensions](#) (std::vector< long > &tileSizes) const  
*Get the current settings of dimension sizes for tiles used in image compression.*
- const String & [name](#) () const  
*return filename of file corresponding to [FITS](#) object*
- [PHDU](#) & [pHDU](#) ()  
*return a reference to the primary [HDU](#).*
- const [PHDU](#) & [pHDU](#) () const  
*return a const reference to the primary [HDU](#).*
- void [read](#) (const std::vector< String > &searchKeys, const std::vector< String > &searchValues, bool readDataFlag=false, const std::vector< String > &hduKeys=std::vector< String >(), int version=1)  
*read method for read header or [HDU](#) that contains specified keywords.*
- void [read](#) (int hduIndex, bool readDataFlag=false, const std::vector< String > &keys=std::vector< String >())  
*read an [HDU](#) specified by index number.*
- void [read](#) (const std::vector< String > &hduNames, const std::vector< std::vector< String > > &keys, bool readDataFlag=false, const std::vector< int > &hduVersions=std::vector< int >())  
*get data from a set of [HDUs](#) from disk file, specifying keys and version numbers.*
- void [read](#) (const std::vector< String > &hduNames, bool readDataFlag=false)  
*get data from a set of [HDUs](#) from disk file.*
- void [read](#) (const String &hduName, bool readDataFlag=false, const std::vector< String > &keys=std::vector< String >(), int version=1)  
*get data from single [HDU](#) from disk file.*
- void [resetPosition](#) ()  
*explicit call to set the fits file pointer to the primary.*
- void [setCompressionType](#) (int compType)

*set the compression algorithm to be used when adding image extensions to the [FITS](#) object.*

- void [setNoiseBits](#) (int noiseBits)  
*Set the cfitsio noisebits parameter used when compressing floating-point images.*
- void [setTileDimensions](#) (const std::vector< long > &tileSizes)  
*Set the dimensions of the tiles into which the image is divided during compression.*

### Static Public Member Functions

- static void [clearErrors](#) ()  
*clear the error stack and set status to zero.*
- static void [setVerboseMode](#) (bool value)  
*set verbose setting for library*
- static bool [verboseMode](#) ()  
*return verbose setting for library*

#### 24.11.1 Detailed Description

Memory object representation of a disk [FITS](#) file. Constructors are provided to get [FITS](#) data from an existing file or to create new [FITS](#) data sets. Overloaded versions allow the user to

a) read from one or more specified extensions, specified by EXTNAME and VERSION or by [HDU](#) number. b either just header information or data on construction c) to specify scalar keyword values to be read on construction d) to open and read an extension that has specified keyword values e) create a new [FITS](#) object and corresponding file, including an empty primary header.

The memory fits object as constructed is always an image of a valid [FITS](#) object, i.e. a primary [HDU](#) is created on construction.

calling the destructor closes the disk file, so that [FITS](#) files are automatically deleted at the end of scope unless other arrangements are made.

## 24.11.2 Constructor & Destructor Documentation

**24.11.2.1 CCfits::FITS::FITS (const String & name, RWmode mode = Read, bool readDataFlag = false, const std::vector< String > & primaryKeys = std::vector<String> ())**

**basic constructor** This basic constructor makes a [FITS](#) object from the given filename. If the mode is Read [default], it will read all of the headers in the file, and all of the data if the readDataFlag is supplied as true. It will also read optional primary keys.

The filename string is passed directly to the cfitsio library: thus the extended filename syntax described in the cfitsio manual should work as documented. (Though the extended file name feature which allows the opening of a particular image located in the row of a table is currently unsupported.) If in Read mode and the extended syntax selects a particular extension, that extension will become the current [HDU](#) position upon construction.

The file name is the only required argument. If the mode is Write and the file does not already exist, a default primary [HDU](#) will be created in the file with BITPIX=8 and NAXIS=0: this mode is designed for writing [FITS](#) files with table extensions only. For files with image data the constructor that specified the data type and number of axes should be called.

### Parameters:

- name* The name of the [FITS](#) file to be read/written
- mode* The read/write mode: must be Read or Write
- readDataFlag* boolean: read data on construction if true
- primaryKeys* Allows optional reading of primary header keys on construction

### Exceptions:

- [NoSuchHDU](#) thrown on [HDU](#) seek error either by index or {name,version}
- [FitsError](#) thrown on non-zero status code from cfitsio when not overridden by [FitsException](#) error to produce more illuminating message.

**24.11.2.2 CCfits::FITS::FITS (const String & name, RWmode mode, const string & hduName, bool readDataFlag = false, const std::vector< String > & hduKeys = std::vector<String> (), const std::vector< String > & primaryKey = std::vector<String> (), int version = 1)**

Open a [FITS](#) file and read a single specified [HDU](#). This and similar constructor variants support reading table data.

Optional arguments allow the reading of primary header keys and specified data from `hduName`, the [HDU](#) to be read. An object representing the primary [HDU](#) is always created: if it contains an image, that image may be read by subsequent calls.

If extended file name syntax is used and selects an extension other than `hduName`, a [FITS::OperationNotSupported](#) exception will be thrown.

**Parameters:**

- name* The name of the [FITS](#) file to be read
- mode* The read/write mode: takes values Read or Write
- hduName* The name of the [HDU](#) to be read.
- hduKeys* Optional array of keywords to be read from the [HDU](#)
- version* Optional version number. If not supplied the first [HDU](#) with name *hduName* is read see above for other parameter definitions

**24.11.2.3 CCfits::FITS::FITS (const String & name, RWmode mode, const std::vector< String > & hduNames, bool readDataFlag = false, const std::vector< String > & primaryKey = std::vector<String>())**

This is intended as a convenience where the file consists of single versions of HDUs and data only, not keys are to be read.

If extended file name syntax is used and selects an extension not listed in `hduNames`, a [FITS::OperationNotSupported](#) exception will be thrown.

**Parameters:**

- hduNames* array of [HDU](#) names to be read.

see above for other parameter definitions.

**24.11.2.4 CCfits::FITS::FITS (const String & fileName, const FITS & source)**

create a new [FITS](#) object and corresponding file with copy of the primary header of the source If the filename corresponds to an existing file and does not start with the '!' character the construction will fail with a [CantCreate](#) exception.

**Parameters:**

- fileName* New file to be created.

*source* A previously created [FITS](#) object to be copied.

see above for other parameter definitions.

**24.11.2.5 CCfits::FITS::FITS (const String & name, RWmode mode, const std::vector< String > & hduNames, const std::vector< std::vector< String > > & hduKeys, bool readDataFlag = false, const std::vector< String > & primaryKeys = std::vector<String>(), const std::vector< int > & hduVersions = std::vector<int>())**

[FITS](#) read constructor in full generality.

**Parameters:**

*hduVersions* an optional version number for each [HDU](#) to be read

*hduKeys* an array of keywords for each [HDU](#) to be read. see above for other parameter definitions.

**24.11.2.6 CCfits::FITS::FITS (const String & name, int bitpix, int naxis, long \* naxes)**

Constructor for creating new [FITS](#) objects containing images. This constructor is only called for creating new files (mode is not an argument) and creates a new primary [HDU](#) with the datatype & axes specified by bitpix, naxis, and naxes. The data are added to the new fits object and file by subsequent calls to [FITS::pHDU\(\).write\( <arguments> \)](#)

A file with a compressed image may be creating by appending to the end of the file name the same "[compress ...]" syntax specified in the cfitsio manual. Note however that the compressed image will be placed in the first extension and NOT in the primary [HDU](#).

If the filename corresponds to an existing file and does not start with the '!' character the construction will fail with a [CantCreate](#) exception.

The arguments are:

**Parameters:**

*name* The file to be written to disk

*bitpix* the datatype of the primary image. *bitpix* may be one of the following CFITSIO constants: BYTE\_IMG, SHORT\_IMG, LONG\_IMG, FLOAT\_IMG,



DOUBLE\_IMG, USHORT\_IMG, ULONG\_IMG. Note that if you send in a *bitpix* of USHORT\_IMG or ULONG\_IMG, CCfits will set [HDU::bitpix\(\)](#) to its signed equivalent (SHORT\_IMG or LONG\_IMG), and then set BZERO to  $2^{15}$  or  $2^{31}$ .

**naxis** the data dimension of the primary image

**naxes** the array of axis lengths for the primary image. Ignored if naxis=0, i.e. the primary header is empty. extensions can be added arbitrarily to the file after this constructor is called. The constructors should write header information to disk:

**24.11.2.7 CCfits::FITS::FITS (const string & name, RWmode mode, int hduIndex, bool readDataFlag = false, const std::vector< String > & hduKeys = std::vector<String>(), const std::vector< String > & primaryKey = std::vector<String>())**

read a single numbered [HDU](#). Constructor analogous to the version that reads by name. This is required since [HDU](#) extensions are not required to have the EXTNAME or HDUNAME keyword by the standard. If there is no name, a dummy name based on the [HDU](#) number is created and becomes the key.

If extended file name syntax is used and selects an extension other than hduIndex, a [FITS::OperationNotSupported](#) exception will be thrown.

#### Parameters:

**hduIndex** The index of the [HDU](#) to be read. see above for other parameter definitions.

**24.11.2.8 CCfits::FITS::FITS (const String & name, RWmode mode, const std::vector< String > & searchKeys, const std::vector< String > & searchValues, bool readDataFlag = false, const std::vector< String > & hduKeys = std::vector<String>(), const std::vector< String > & primaryKey = std::vector<string>(), int version = 1)**

open fits file and read [HDU](#) which contains supplied keywords with [optional] specified values (sometimes one just wants to know that the keyword is present). Optional parameters allows the reading of specified primary [HDU](#) keys and specified columns and keywords in the [HDU](#) of interest.

**Parameters:**

- name*** The name of the [FITS](#) file to be read
- mode*** The read/write mode: must be Read or Write
- searchKeys*** A string vector of keywords to search for in each header
- searchValues*** A string vector of values those keywords are required to have for success. Note that the keys must be of type string. If any value does not need to be checked the corresponding searchValue element can be empty.
- readDataFlag*** boolean: if true, read data if [HDU](#) is found
- hduKeys*** Allows optional reading of keys in the [HDU](#) that is searched for if it is successfully found
- primaryKey*** Allows optional reading of primary header keys on construction
- version*** Optional version number. If specified, checks the EXTVERS keyword.

**Exceptions:**

- FitsError*** thrown on non-zero status code from cfitsio when not overridden by [FitsException](#) error to produce more illuminating message.

**24.11.3 Member Function Documentation****24.11.3.1 void CCfits::FITS::addImage (const String & hduName, int bpix, std::vector< long > & naxes, int version = 1)**

Add an image extension to an existing [FITS](#) object. (File with w or rw access). Does not make primary images, which are built in the constructor for the [FITS](#) file. The image data is not added here: it can be added by a call to one of the [ExtHDU::write](#) functions.

*bpix* may be one of the following CFITSIO constants: BYTE\_IMG, SHORT\_IMG, LONG\_IMG, FLOAT\_IMG, DOUBLE\_IMG, USHORT\_IMG, ULONG\_IMG. Note that if you send in a *bpix* of USHORT\_IMG or ULONG\_IMG, CCfits will set [HDU::bitpix\(\)](#) to its signed equivalent (SHORT\_IMG or LONG\_IMG), and then set BZERO to  $2^{15}$  or  $2^{31}$ .

**Todo**

- Add a function for replacing the primary image

**24.11.3.2** `Table * CCfits::FITS::addTable (const String & hduName, int rows, const std::vector< String > & columnName = std::vector<String> (), const std::vector< String > & columnFmt = std::vector<String> (), const std::vector< String > & columnUnit = std::vector<String> (), HduType type = BinaryTbl, int version = 1)`

Add a table extension to an existing [FITS](#) object. Add extension to [FITS](#) object for file with w or rw access.

**Parameters:**

*rows* The number of rows in the table to be created.

*columnName* A vector containing the table column names

*columnFmt* A vector containing the table column formats

*columnUnit* (Optional) a vector giving the units of the columns.

*type* The table type - AsciiTbl or BinaryTbl (defaults to BinaryTbl) the lists of columns are optional - one can create an empty table extension but if supplied, colType, columnName and colFmt must have equal dimensions.

**Todo**

the code should one day check that the version keyword is higher than any other versions already added to the [FITS](#) object (although cfitsio doesn't do this either).

**24.11.3.3** `void CCfits::FITS::copy (const HDU & source)`

copy the [HDU](#) source into the [FITS](#) object. This function adds a copy of an [HDU](#) from another file into \*this. It does not create a duplicate of an [HDU](#) in the file associated with \*this.

**24.11.3.4** `const String & CCfits::FITS::currentExtensionName () const`

return the name of the extension that the fitsfile is currently addressing. If the extension in question does not have an EXTNAME or HDUNAME keyword, then the function returns \$HDU\$n, where n is the sequential [HDU](#) index number (primary [HDU](#) = 0).

**24.11.3.5 void CCfits::FITS::deleteExtension (int *doomed*)**

Delete extension specified by extension number. This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

**24.11.3.6 void CCfits::FITS::deleteExtension (const String & *doomed*, int *version* = 1)**

Delete extension specified by name and version number. Removes extension from [FITS](#) object and memory copy.

**Parameters:**

*doomed* the name of the extension to be deleted

*version* an optional version number, the EXTVER keyword, defaults to 1

**Exceptions:**

[\*NoSuchHDU\*](#) Thrown if there is no extension with the specified version number

[\*FitsError\*](#) Thrown if there is a non-zero status code from cfitsio, e.g. if the delete operation is applied to a [FITS](#) file opened for read only access.

**24.11.3.7 void CCfits::FITS::destroy () throw ()**

Erase [FITS](#) object and close corresponding file. Force deallocation and erase of elements of a [FITS](#) memory object. Allows a reset of everything inside the [FITS](#) object, and closes the file. The object is inaccessible after this call.

destroy is public to allow users to reuse a symbol for a new file, but it is identical in operation to the destructor.

**24.11.3.8 const std::multimap< std::String, ExtHDU \* > & CCfits::FITS::extension () const**

return const reference to the extension container This is useful for such operations as [extension\(\).size\(\)](#) etc.

### 24.11.3.9 Table & CCfits::FITS::filter (const String & *expression*, ExtHDU & *inputTable*, bool *overwrite* = `true`, bool *readData* = `false`)

Filter the rows of the *inputTable* with the condition *expression*, and return a reference to the resulting [Table](#). This function provides an object oriented version of `cfitsio's fits_select_rows` call. The *expression* string is any boolean expression involving the names of the columns in the input table (e.g., if there were a column called "density", a valid expression might be "DENSITY > 3.5": see the `cfitsio` documentation for further details).

[N.B. the "append" functionality described below does not work when linked with `cfitsio 2.202` or prior because of a known issue with that version of the library. This causes the output to be a new extension with a correct header copy and version number but without the filtered data]. If the *inputTable* is an Extension HDU of this [FITS](#) object, then if *overwrite* is `true` the operation will overwrite the *inputTable* with the filtered version, otherwise it will append a new HDU with the same extension name but the next highest version (EXTVER) number available.

### 24.11.3.10 void CCfits::FITS::flush ()

flush buffer contents to disk Provides manual control of disk writing operation. Image data are flushed automatically to disk after the write operation is completed, but not column data.

### 24.11.3.11 void CCfits::FITS::getTileDimensions (std::vector< long > & *tileSizes*) const

Get the current settings of dimension sizes for tiles used in image compression.

#### Parameters:

*tileSizes* A vector to be filled with `cfitsio's` current tile dimension settings. `CCfits` will resize this vector to contain the proper number of values.

### 24.11.3.12 CCfits::FITS::read (const std::vector< String > & *searchKeys*, const std::vector< String > & *searchValues*, bool *readDataFlag* = `false`, const std::vector< String > & *hduKeys* = `std::vector<String>()`, int *version* = 1)

read method for read header or [HDU](#) that contains specified keywords.

**Parameters:**

- searchKeys* A string vector of keywords to search for in each header
- searchValues* A string vector of values those keywords are required to have for success. Note that the keys must be of type string. If any value does not need to be checked the corresponding searchValue element can be empty.
- readDataFlag* boolean: if true, read data if [HDU](#) is found
- hduKeys* Allows optional reading of keys in the [HDU](#) that is searched for if it is successfully found
- version* Optional version number. If specified, checks the EXTVERS keyword.

**24.11.3.13** `void CCfits::FITS::read (int hduIndex, bool readDataFlag = false, const std::vector< String > & keys = std::vector<String>())`

read an [HDU](#) specified by index number. This is provided to allow reading of HDUs after construction. see above for parameter definitions.

**24.11.3.14** `void CCfits::FITS::read (const std::vector< String > & hduNames, const std::vector< std::vector< String > > & keys, bool readDataFlag = false, const std::vector< int > & hduVersions = std::vector<int>())`

get data from a set of HDUs from disk file, specifying keys and version numbers. This is provided to allow reading of HDUs after construction. see above for parameter definitions.

**24.11.3.15** `void CCfits::FITS::read (const std::vector< String > & hduNames, bool readDataFlag = false)`

get data from a set of HDUs from disk file. This is provided to allow reading of HDUs after construction. see above for parameter definitions.

**24.11.3.16** `void CCfits::FITS::read (const String & hduName, bool readDataFlag = false, const std::vector< String > & keys = std::vector<String> (), int version = 1)`

get data from single HDU from disk file. This is provided to allow the adding of additional HDUs to the FITS object after construction of the FITS object. After the read() functions have been called for the FITS object, subsequent read method to the Primary, ExtHDU, and Column objects will retrieve data from the FITS object in memory (those methods can be called to read data in those HDU objects that was not read when the HDU objects were constructed).

All the read functions will throw NoSuchHDU exceptions on seek errors since they involve constructing HDU objects.

The parameter definitions are as documented for the corresponding constructor.

**24.11.3.17** `void CCfits::FITS::setCompressionType (int compType)`

set the compression algorithm to be used when adding image extensions to the FITS object.

**Parameters:**

*compType* Currently 3 symbolic constants are defined in cfitsio for specifying compression algorithms: GZIP\_1, RICE\_1, and PLIO\_1. See the cfitsio documentation for more information about these algorithms. Entering NULL for compType will turn off compression and cause normal FITS images to be written.

**24.11.3.18** `void CCfits::FITS::setNoiseBits (int noiseBits)`

Set the cfitsio noisebits parameter used when compressing floating-point images. The default value is 4. Decreasing the value of noisebits will improve the overall compression efficiency at the expense of losing more information.

**24.11.3.19** `void CCfits::FITS::setTileDimensions (const std::vector< long > & tileSizes)`

Set the dimensions of the tiles into which the image is divided during compression.

**Parameters:**

*tileSizes* A vector of length N containing the tile dimesions. If N is less than the number of dimensions of the image it is applied to, the unspecified dimensions will be assigned a size of 1 pixel. If N is larger than the number of image dimensions, the extra dimensions will be ignored.

The default cfitsio behavior is to create tiles with dimensions NAXIS1 x 1 x 1 etc. up to the number of image dimensions.

**24.11.3.20 bool CCfits::FITS::verboseMode () [inline, static]**

return verbose setting for library If true, all messages that are reported by exceptions are printed to std::cerr.

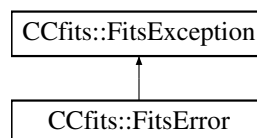
The documentation for this class was generated from the following files:

- FITS.h
- FITS.cxx

**24.12 CCfits::FitsError Class Reference**

[FitsError](#) is the exception thrown by non-zero cfitsio status codes.

#include <FitsError.h> Inheritance diagram for CCfits::FitsError::

**Public Member Functions**

- [FitsError](#) (int errornum, bool silent=true)  
*ctor for cfitsio exception: translates status code into cfitsio error message*

**24.12.1 Detailed Description**

[FitsError](#) is the exception thrown by non-zero cfitsio status codes.



## 24.12.2 Constructor & Destructor Documentation

### 24.12.2.1 CCfits::FitsError::FitsError (int *errornum*, bool *silent* = true)

ctor for cfitsio exception: translates status code into cfitsio error message The exception prefixes the string "Fits Error: " to the message printed by cfitsio.

#### Parameters:

- errornum* The cfitsio status code produced by the error.
- silent* A boolean controlling the printing of messages

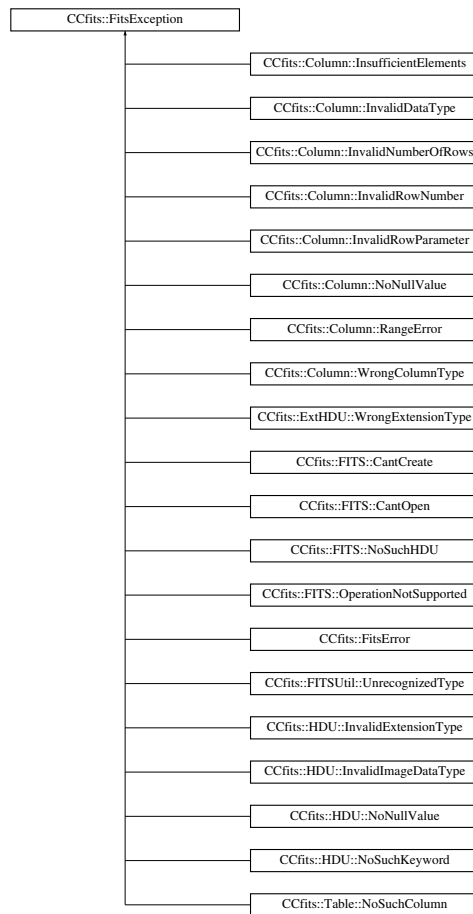
The documentation for this class was generated from the following files:

- FitsError.h
- FitsError.cxx

## 24.13 CCfits::FitsException Class Reference

[FitsException](#) is the base class for all exceptions thrown by this library.

#include <FitsError.h>Inheritance diagram for CCfits::FitsException::



## Public Member Functions

- [FitsException](#) (const string &msg, bool &silent)
- const string & [message](#) () const

*returns the error message*

### 24.13.1 Detailed Description

[FitsException](#) is the base class for all exceptions thrown by this library. All exceptions derived from this class can be caught by a single 'catch' clause catching [FitsException](#) by reference (which is the point of this base class design).

A static "verboseMode" parameter is provided by the [FITS](#) class to control diagnostics

- if `FITS::verboseMode()` is true, all diagnostics are printed (for debugging purposes). If not, then a boolean *silent* determines printing of messages. Each exception derived from `FitsException` must define a default value for the *silent* parameter.

### 24.13.2 Constructor & Destructor Documentation

#### 24.13.2.1 CCfits::FitsException::FitsException (const string & *diag*, bool & *silent*)

##### Parameters:

- diag* A diagnostic string to be printed optionally.
- silent* A boolean controlling the printing of messages

### 24.13.3 Member Function Documentation

#### 24.13.3.1 const string & CCfits::FitsException::message () const [inline]

returns the error message This returns the diagnostic error message associated with the exception object, and which is accessible regardless of the verboseMode and silent flag settings.

The documentation for this class was generated from the following files:

- FitsError.h
- FitsError.cxx

## 24.14 CCfits::FitsFatal Class Reference

[potential] base class for exceptions to be thrown on internal library error.

```
#include <FitsError.h>
```

### Public Member Functions

- `FitsFatal` (const string &*diag*)  
*Prints a message starting "\*\*\* CCfits Fatal Error: ..." and calls terminate().*

### 24.14.1 Detailed Description

[potential] base class for exceptions to be thrown on internal library error. As of this version there are no subclasses. This error requests that the user reports this circumstance to HEASARC.

### 24.14.2 Constructor & Destructor Documentation

#### 24.14.2.1 CCfits::FitsFatal::FitsFatal (const string & *diag*)

Prints a message starting "\*\*\* CCfits Fatal Error: ..." and calls *terminate()*.

#### Parameters:

*diag* A diagnostic string to be printed identifying the context of the error.

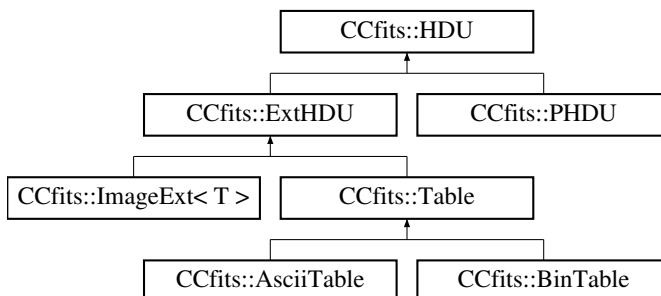
The documentation for this class was generated from the following files:

- FitsError.h
- FitsError.cxx

## 24.15 CCfits::HDU Class Reference

Base class for all [HDU](#) [Header-Data Unit] objects.

#include <HDU.h> Inheritance diagram for CCfits::HDU::



### Classes

- class [InvalidExtensionType](#)

*exception to be thrown if user requests extension type that can not be understood as [ImageExt](#), [AsciiTable](#) or [BinTable](#).*

- class [InvalidImageDataType](#)  
*exception to be thrown if user requests creation of an image of type not supported by cfitsio.*
- class [NoNullValue](#)  
*exception to be thrown on seek errors for keywords.*
- class [NoSuchKeyword](#)  
*exception to be thrown on seek errors for keywords.*

### Public Member Functions

- [HDU](#) (const [HDU](#) &right)  
*copy constructor*
- [Keyword](#) \* [addKey](#) (const [Keyword](#) \*inKeyword)  
*create a copy of an existing [Keyword](#) and add to [HDU](#)*
- template<typename T >  
[Keyword](#) & [addKey](#) (const String &name, T val, const String &comment)  
*create a new keyword in the [HDU](#) with specified value and comment fields*
- long [axes](#) () const  
*return the number of axes in the [HDU](#) data section (always 2 for tables).*
- long [axis](#) (size\_t index) const  
*return the size of axis numbered index [zero based].*
- long [bitpix](#) () const  
*return the data type keyword.*
- virtual [HDU](#) \* [clone](#) (FITSBase \*p) const =0  
*virtual copy constructor, to be implemented in subclasses.*
- const string & [comment](#) () const  
*return the comment string previously read by [getComment\(\)](#)*
- void [copyAllKeys](#) (const [HDU](#) \*inHdu)  
*copy all keys from another header*

- void `deleteKey` (const String &doomed)  
*delete a keyword from the header*
- fitsfile \* `fitsPointer` () const  
*return the fitsfile pointer for the `FITS` object containing the `HDU`*
- std::pair< unsigned long, unsigned long > `getChecksum` () const  
*compute and return the checksum values for the `HDU` without creating or modifying the `CHECKSUM/DATASUM` keywords.*
- const String & `getComments` ()  
*read the comments from the `HDU` and add it to the `FITS` object.*
- const String & `getHistory` ()  
*read the history information from the `HDU` and add it to the `FITS` object.*
- const string & `history` () const  
*return the history string previously read by `getHistory()`*
- int `index` () const  
*return the `HDU` number*
- void `index` (int value)  
*set the `HDU` number*
- const `Keyword` & `keyWord` (const string &keyname) const  
*return a (previously read) keyword from the `HDU` object. const version*
- const std::map< string, `Keyword` \* > & `keyWord` () const  
*return the associative array containing the `HDU` Keywords that have been read so far.*
- `Keyword` & `keyWord` (const String &keyName)  
*return a (previously read) keyword from the `HDU` object.*
- std::map< String, `Keyword` \* > & `keyWord` ()  
*return the associative array containing the `HDU` keywords so far read.*
- virtual void `makeThisCurrent` () const  
*move the fitsfile pointer to this current `HDU`.*
- bool `operator!=` (const `HDU` &right) const  
*inequality operator*

- bool `operator==` (const [HDU](#) &right) const  
*equality operator*
- FITSBase \* `parent` () const  
*return reference to the pointer representing the FITSBase object containing the [HDU](#)*
- void `readAllKeys` ()  
*read all of the keys in the header*
- template<typename T >  
void `readKey` (const String &keyName, T &val)  
*read a keyword of specified type from the header of a disk [FITS](#) file and return its value.*
- template<typename T >  
void `readKeys` (std::vector< String > &keyNames, std::vector< T > &vals)  
*read a set of specified keywords of the same data type from the header of a disk [FITS](#) file and return their values*
- virtual void `scale` (double value)  
*set the BSCALE keyword value for images (see warning for images of int type)*
- virtual double `scale` () const  
*return the BSCALE keyword value*
- void `suppressScaling` (bool toggle=true)  
*turn off image scaling regardless of the BSCALE and BZERO keyword values*
- void `updateChecksum` ()  
*update the CHECKSUM keyword value, assuming DATASUM exists and is correct*
- std::pair< int, int > `verifyChecksum` () const  
*verify the [HDU](#) by computing the checksums and comparing them with the CHECKSUM/DATASUM keywords*
- void `writeChecksum` ()  
*compute and write the DATASUM and CHECKSUM keyword values*
- void `writeComment` (const String &comment="Generic Comment")  
*write a comment string.*
- void `writeDate` ()

*write a date string to \*this.*

- void [writeHistory](#) (const String &history="Generic History String")  
*write a history string.*
- virtual void [zero](#) (double value)  
*set the BZERO keyword value for images (see warning for images of int type)*
- virtual double [zero](#) () const  
*return the BZERO keyword value*

### Static Public Member Functions

- static std::vector< int > [keywordCategories](#) ()  
*return the enumerated keyword categories used by [readAllKeys\(\)](#) and [copyAllKeys\(\)](#)*

### Protected Member Functions

- [HDU](#) (FITSBase \*p, int bitpix, int naxis, const std::vector< long > &axes)  
*constructor for creating new [HDU](#) objects, called by [HDU](#) subclasses writing to [FITS](#) files.*
- [HDU](#) (FITSBase \*p=0)  
*default constructor, called by [HDU](#) subclasses that read from [FITS](#) files.*
- virtual [~HDU](#) ()  
*destructor*
- std::vector< long > & [naxes](#) ()  
*return the [HDU](#) data axis array.*

#### 24.15.1 Detailed Description

Base class for all [HDU](#) [Header-Data Unit] objects. [HDU](#) objects in CCfits are either [PHDU](#) (Primary [HDU](#) objects) or [ExtHDU](#) (Extension [HDU](#)) objects. Following the behavior. ExtHDUs are further subclassed into [ImageExt](#) or [Table](#) objects, which are finally [AsciiTable](#) or [BinTable](#) objects.



HDU's public interface gives access to properties that are common to all HDUs, largely required keywords, and functions that are common to all HDUs, principally the manipulation of keywords and their values.

HDUs must be constructed by HDUCreator objects which are called by [FITS](#) methods. Each [HDU](#) has an embedded pointer to a FITSBase object, which is private to [FITS](#) [FITSBase is a pointer encapsulating the resources of [FITS](#). For details of this coding idiom see Exceptional C++ by Herb Sutter (2000) and references therein].

### 24.15.2 Member Function Documentation

#### 24.15.2.1 Keyword \* CCfits::HDU::addKey (const Keyword \* *inKeyword*)

create a copy of an existing [Keyword](#) and add to [HDU](#) This is particularly useful for copying Keywords from one [HDU](#) to another. For example the *inKeyword* pointer might come from a different HDU's `std::map<string,Keyword*>`. If a keyword with this name already exists, it will be overwritten. The return value is a pointer to the newly created [Keyword](#) inserted into this [HDU](#). Also see [copyAllKeys\(\)](#).

#### 24.15.2.2 `template<typename T > Keyword & CCfits::HDU::addKey (const String & name, T value, const String & comment) [inline]`

create a new keyword in the [HDU](#) with specified value and comment fields The function returns a reference to keyword object just created. If a keyword with this name already exists, it will be overwritten. Note that this is mostly intended for adding user-defined keywords. It should not be used to add keywords for which there are already specific [HDU](#) functions, such as scaling or checksum. Nor should it be used for image or column structural keywords, such as BITPIX, NAXIS, TFORN, etc. As a general rule, it is best to use this for keywords belonging to the same categories listed in the [keywordCategories\(\)](#) function.

Parameters:

#### Parameters:

***name*** (String) The keyword name

***value*** (Recommended T = String, double, `std::complex<float>`, int, or bool

***comment*** (String) the keyword value

It is possible to create a keyword with a value of any of the allowed data types in fitsio (see the cfitsio manual section 4.3). However one should be aware that if this keyword value is read in from the file at a later time, it will be stored in a templated [Keyword](#)

subclass (`KeyData<T>`) where `T` will be one of the recommended types listed above. Also see [Keyword::value](#) (`T& val`) for more details.

#### 24.15.2.3 `long CCfits::HDU::axis (size_t index) const [inline]`

return the size of axis numbered index [zero based]. return the length of [HDU](#) data axis `i`.

#### 24.15.2.4 `long CCfits::HDU::bitpix () const [inline]`

return the data type keyword. Takes values denoting the image data type for images, and takes the fixed value 8 for tables.

#### 24.15.2.5 `void CCfits::HDU::copyAllKeys (const HDU * inHdu)`

copy all keys from another header Parameters:

##### Parameters:

***inHdu*** (const HDU\*) An existing [HDU](#) whose keys will be copied.

This will copy all keys that exist in the `keyWord` map of *inHdu*, and which belong to one of the keyword classes returned by the [keywordCategories\(\)](#) function. This is the same group of keyword classes used by [readAllKeys\(\)](#).

#### 24.15.2.6 `void CCfits::HDU::deleteKey (const String & doomed)`

delete a keyword from the header removes *doomed* from the [FITS](#) file and from the [FITS](#) object

#### 24.15.2.7 `std::pair< unsigned long, unsigned long > CCfits::HDU::getChecksum () const`

compute and return the checksum values for the [HDU](#) without creating or modifying the CHECKSUM/DATASUM keywords. Wrapper for the CFITSIO function `fits_get_`

chksum: This returns a `std::pair<unsigned long, unsigned long>` where the pair's first data member holds the datasum value and second holds the hdusum value.

#### 24.15.2.8 `const String & CCfits::HDU::getComments ()`

read the comments from the [HDU](#) and add it to the [FITS](#) object. The comment string found in the header is concatenated and returned to the calling function

#### 24.15.2.9 `const String & CCfits::HDU::getHistory ()`

read the history information from the [HDU](#) and add it to the [FITS](#) object. The history string found in the header is concatenated and returned to the calling function

#### 24.15.2.10 `static std::vector< int > CCfits::HDU::keywordCategories ()` `[static]`

return the enumerated keyword categories used by [readAllKeys\(\)](#) and [copyAllKeys\(\)](#). This returns a vector of integers indicating which categories of keywords apply for the [readAllKeys](#) and [copyAllKeys](#) functions. The list of categories currently hardcoded is: `TYP_CMPRS_KEY` (20), `TYP_CKSUM_KEY` (100), `TYP_WCS_KEY` (110), `TYP_REFSYS_KEY` (120), and `TYP_USER_KEY` (150).

For the list of ALL keyword categories, see the CFITSIO documentation for the `fits_get_keyclass` function.

#### 24.15.2.11 `void CCfits::HDU::makeThisCurrent () const` `[virtual]`

move the fitsfile pointer to this current [HDU](#). This function should never need to be called by the user since it is called internally whenever required.

Reimplemented in [CCfits::ExtHDU](#).

#### 24.15.2.12 `void CCfits::HDU::readAllKeys ()`

read all of the keys in the header This member function reads keys that are not meta data for columns or image information, [which are considered to be part of the column or

image objects]. Also, history and comment keys are read and returned by [getHistory\(\)](#) and [getComment\(\)](#). The exact list of keyword classes this will read is returned by the function [keywordCategories\(\)](#).

Note that `readAllKeys` can only construct keys of type string, double, `complex<float>`, integer, and bool because the [FITS](#) header records do not encode exact type information.

#### 24.15.2.13 `template<typename T> void CCfits::HDU::readKey (const String & keyName, T & val) [inline]`

read a keyword of specified type from the header of a disk [FITS](#) file and return its value. T is one of the types String, double, float, int, `std::complex<float>`, and bool. If a [Keyword](#) object with the name *keyName* already exists in this [HDU](#) due to a previous read call, then this will re-read from the file and create a new [Keyword](#) object to replace the existing one.

#### 24.15.2.14 `template<typename T> void CCfits::HDU::readKeys (std::vector<String> & keyNames, std::vector< T> & vals) [inline]`

read a set of specified keywords of the same data type from the header of a disk [FITS](#) file and return their values T is one of the types String, double, float, int, `std::complex<float>`, and bool.

#### 24.15.2.15 `void CCfits::HDU::scale (double value) [inline, virtual]`

set the BSCALE keyword value for images (see warning for images of int type) For primary HDUs and image extensions, this will add (or update) the BSCALE keyword in the header. The new setting will affect future image array read/writes as described in section 4.7 Data Scaling of the CFITSIO manual. For table extensions this function does nothing.

WARNING: If the image contains **integer-type data** (as indicated by the [bitpix\(\)](#) return value), the new scale and zero value combination must not be such that the scaled data would require a floating-point type (this uses the CFITSIO function `fits_get_img_equivtype` to make the determination). If this situation occurs, the function will throw a [FitsException](#).

Reimplemented in [CCfits::ImageExt< T>](#), and [CCfits::PHDU](#).

**24.15.2.16 void CCfits::HDU::suppressScaling (bool *toggle* = true)**

turn off image scaling regardless of the BSCALE and BZERO keyword values For *toggle* = true, this turns off image scaling for future read/writes by resetting the scale and zero to 1.0 and 0.0 respectively. It does NOT modify the BSCALE and BZERO keywords. If *toggle* = false, the scale and zero values will be restored to the keyword values.

**24.15.2.17 void CCfits::HDU::updateChecksum ()**

update the CHECKSUM keyword value, assuming DATASUM exists and is correct Wrapper for the CFITSIO function fits\_update\_chksum: This recomputes and writes the CHECKSUM value with the assumption that the DATASUM value is correct. If the DATASUM keyword doesn't yet exist or is not up-to-date, use the [HDU::writeChecksum](#) function instead. This will throw a [FitsError](#) exception if called when there is no DATASUM keyword in the header.

**24.15.2.18 std::pair< int, int > CCfits::HDU::verifyChecksum () const**

verify the [HDU](#) by computing the checksums and comparing them with the CHECKSUM/DATASUM keywords Wrapper for the CFITSIO function fits\_verify\_chksum: The data unit is verified correctly if the computed checksum equals the DATASUM keyword value, and the [HDU](#) is verified if the entire checksum equals zero (see the CFITSIO manual for further details).

This returns a std::pair<int,int> where the pair's first data member = DATAOK and second = HDUOK. DATAOK and HDUOK values will be = 1 if verified correctly, 0 if the keyword is missing, and -1 if the computed checksum is not correct.

**24.15.2.19 void CCfits::HDU::writeChecksum ()**

compute and write the DATASUM and CHECKSUM keyword values Wrapper for the CFITSIO function fits\_write\_chksum: This performs the datasum and checksum calculations for this [HDU](#), as described in the CFITSIO manual. If either the DATASUM or CHECKSUM keywords already exist, their values will be updated.

**24.15.2.20** `void CCfits::HDU::writeComment (const String & comment = "Generic Comment")`

write a comment string. A default value for the string is given ("Generic Comment String") so users can put a placeholder call to this function in their code.

**24.15.2.21** `void CCfits::HDU::writeHistory (const String & history = "Generic History String")`

write a history string. A default value for the string is given ("Generic History String") so users can put a placeholder call to this function in their code.

**24.15.2.22** `void CCfits::HDU::zero (double value) [inline, virtual]`

set the BZERO keyword value for images (see warning for images of int type) For primary HDUs and image extensions, this will add (or update) the BZERO keyword in the header. The new setting will affect future image array read/writes as described in section 4.7 Data Scaling of the CFITSIO manual. For table extensions this function does nothing.

WARNING: If the image contains **integer-type data** (as indicated by the [bitpix\(\)](#) return value), the new scale and zero value combination must not be such that the scaled data would require a floating-point type (this uses the CFITSIO function `fits_get_img_equivtype` to make the determination). If this situation occurs, the function will throw a [FitsException](#).

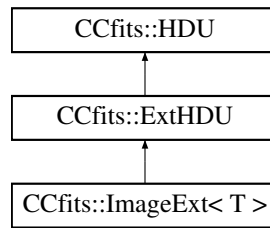
Reimplemented in [CCfits::ImageExt< T >](#), and [CCfits::PHDU](#).

The documentation for this class was generated from the following files:

- HDU.h
- HDU.cxx

## 24.16 CCfits::ImageExt< T > Class Template Reference

Inheritance diagram for CCfits::ImageExt< T >::



### Public Member Functions

- virtual `~ImageExt()`  
*destructor*
- virtual `ImageExt< T > * clone(FITSBase *p) const`  
*virtual copy constructor*
- `const std::valarray< T > & image() const`  
*return the image data*
- virtual `void readData(bool readFlag=false, const std::vector< String > &keys=std::vector< String >())`  
*read Image extension HDU data*
- virtual `double scale() const`  
*return the BSCALE keyword value*
- virtual `void scale(double value)`  
*set the BSCALE keyword value for images (see warning for images of int type)*
- virtual `double zero() const`  
*return the BZERO keyword value*
- virtual `void zero(double value)`  
*set the BZERO keyword value for images (see warning for images of int type)*

#### 24.16.1 Detailed Description

`template<typename T> class CCfits::ImageExt< T >`

`ImageExt<T>` is a subclass of `ExtHDU` that contains image data of type `T`.

### 24.16.2 Member Function Documentation

**24.16.2.1** `template<typename T> void CCfits::ImageExt< T >::readData  
(bool readFlag = false, const std::vector< String > & keys =  
std::vector<String>()) [inline, virtual]`

read Image extension [HDU](#) data Called by [FITS](#) ctor, not intended for general use.  
parameters control how much gets read on initialization.

**Parameters:**

*readFlag* read the image data if true

*key* a vector of strings of keyword names to be read from the [HDU](#)

Implements [CCfits::ExtHDU](#).

**24.16.2.2** `template<typename T> void CCfits::ImageExt< T >::scale (double  
value) [inline, virtual]`

set the BSCALE keyword value for images (see warning for images of int type) For  
primary HDUs and image extensions, this will add (or update) the BSCALE keyword  
in the header. The new setting will affect future image array read/writes as described  
in section 4.7 Data Scaling of the CFITSIO manual. For table extensions this function  
does nothing.

WARNING: If the image contains **integer-type data** (as indicated by the [bitpix\(\)](#) return  
value), the new scale and zero value combination must not be such that the scaled data  
would require a floating-point type (this uses the CFITSIO function `fits_get_img_  
equivtype` to make the determination). If this situation occurs, the function will throw  
a [FitsException](#).

Reimplemented from [CCfits::HDU](#).

**24.16.2.3** `template<typename T> void CCfits::ImageExt< T >::zero (double  
value) [inline, virtual]`

set the BZERO keyword value for images (see warning for images of int type) For  
primary HDUs and image extensions, this will add (or update) the BZERO keyword  
in the header. The new setting will affect future image array read/writes as described  
in section 4.7 Data Scaling of the CFITSIO manual. For table extensions this function  
does nothing.



WARNING: If the image contains **integer-type data** (as indicated by the [bitpix\(\)](#) return value), the new scale and zero value combination must not be such that the scaled data would require a floating-point type (this uses the CFITSIO function `fits_get_img_equivtype` to make the determination). If this situation occurs, the function will throw a [FitsException](#).

Reimplemented from [CCfits::HDU](#).

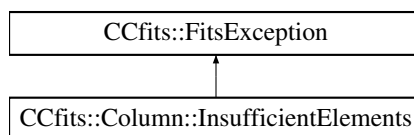
The documentation for this class was generated from the following file:

- ImageExt.h

## 24.17 CCfits::Column::InsufficientElements Class Reference

Exception thrown if the data supplied for a write operation is less than declared.

```
#include <Column.h>
Inheritance diagram for CCfits::Column::InsufficientElements::
```



### Public Member Functions

- [InsufficientElements](#) (const String &msg, bool silent=true)

*Exception ctor, prefixes the string "FitsError: not enough elements supplied for write operation: " before the specific message.*

#### 24.17.1 Detailed Description

Exception thrown if the data supplied for a write operation is less than declared. This circumstance generates an exception to avoid unexpected behaviour after the write operation is completed. It can be avoided by resizing the input array appropriately.

#### 24.17.2 Constructor & Destructor Documentation

##### 24.17.2.1 CCfits::Column::InsufficientElements::InsufficientElements (const String &msg, bool silent = true)

Exception ctor, prefixes the string "FitsError: not enough elements supplied for write operation: " before the specific message.

#### Parameters:


- msg* A specific diagnostic message, usually the column name
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

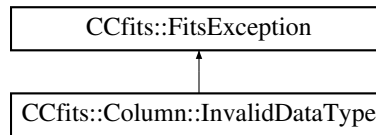
The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

## 24.18 CCfits::Column::InvalidDataType Class Reference

Exception thrown for invalid data type inputs.

#include <Column.h> **Inheritance**  **for** **CC-**  
fits::Column::InvalidDataType::



#### Public Member Functions

- [InvalidDataType](#) (const String &str=string(), bool silent=true)  
*Exception ctor, prefixes the string "FitsError: Incorrect data type: " before the specific message.*

#### 24.18.1 Detailed Description

Exception thrown for invalid data type inputs. This exception is thrown if the user requests an implicit data type conversion to a datatype that is not one of the supported types (see fitsio.h for details).

## 24.18.2 Constructor & Destructor Documentation

### 24.18.2.1 CCfits::Column::InvalidDataType::InvalidDataType (const String & *str* = `string()`, bool *silent* = `true`)

Exception ctor, prefixes the string "FitsError: Incorrect data type: " before the specific message.

#### Parameters:

- str* A specific diagnostic message
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

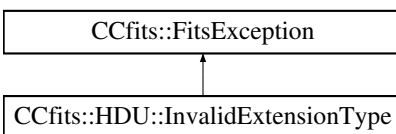
The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

## 24.19 CCfits::HDU::InvalidExtensionType Class Reference

exception to be thrown if user requests extension type that can not be understood as [ImageExt](#), [AsciiTable](#) or [BinTable](#).

#include <Hdu.h> Inheritance diagram for CCfits::HDU::InvalidExtensionType:



### Public Member Functions

- [InvalidExtensionType](#) (const string &diag, bool silent=true)  
*Exception ctor, prefixes the string "Fits Error: Extension Type: " before the specific message.*

### 24.19.1 Detailed Description

exception to be thrown if user requests extension type that can not be understood as [ImageExt](#), [AsciiTable](#) or [BinTable](#).

## 24.19.2 Constructor & Destructor Documentation

### 24.19.2.1 CCfits::HDU::InvalidExtensionType::InvalidExtensionType (const string &diag, bool silent = true)

Exception ctor, prefixes the string "Fits Error: Extension Type: " before the specific message.

#### Parameters:

- diag* A specific diagnostic message
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

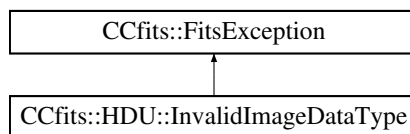
The documentation for this class was generated from the following files:

- HDU.h
- HDU.cxx

## 24.20 CCfits::HDU::InvalidImageDataType Class Reference

exception to be thrown if user requests creation of an image of type not supported by cfitsio.

#include <HDU.h>Inheritance diagram for CCfits::HDU::InvalidImageDataType::



### Public Member Functions

- [InvalidImageDataType](#) (const string &diag, bool silent=true)  
*Exception ctor, prefixes the string "Fits Error: Invalid Data Type for Image " before the specific message.*

### 24.20.1 Detailed Description

exception to be thrown if user requests creation of an image of type not supported by cfitsio.

## 24.20.2 Constructor & Destructor Documentation

### 24.20.2.1 CCfits::HDU::InvalidImageDataType::InvalidImageDataType (const string & *diag*, bool *silent* = true)

Exception ctor, prefixes the string "Fits Error: Invalid Data Type for Image " before the specific message.

#### Parameters:

- diag* A specific diagnostic message
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

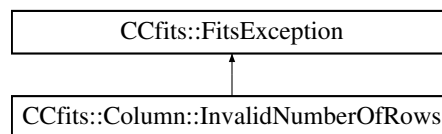
The documentation for this class was generated from the following files:

- HDU.h
- HDU.cxx

## 24.21 CCfits::Column::InvalidNumberOfRows Class Reference

Exception thrown if user enters a non-positive number for the number of rows to write.

#include <Column.h> Inheritance diagram for CCfits::Column::InvalidNumberOfRows::



### Public Member Functions

- [InvalidNumberOfRows](#) (size\_t number, bool silent=true)  
*Exception ctor, prefixes the string "Fits Error: number of rows to write must be positive " before the specific message.*

### 24.21.1 Detailed Description

Exception thrown if user enters a non-positive number for the number of rows to write.

### 24.21.2 Constructor & Destructor Documentation

#### 24.21.2.1 CCfits::Column::InvalidNumberOfRows::InvalidNumberOfRows (size\_t *number*, bool *silent* = true)

Exception ctor, prefixes the string "Fits Error: number of rows to write must be positive " before the specific message.

##### Parameters:

- number* The number of rows entered.
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

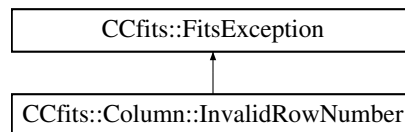
The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

## 24.22 CCfits::Column::InvalidRowNumber Class Reference

Exception thrown on attempting to read a row number beyond the end of a table.

#include <Column.h> Inheritance diagram for CCfits::Column::InvalidRowNumber::



### Public Member Functions

- [InvalidRowNumber](#) (const String &diag, bool silent=true)  
*Exception ctor, prefixes the string "FitsError: Invalid Row Number - Column: " before the specific message.*

#### 24.22.1 Detailed Description

Exception thrown on attempting to read a row number beyond the end of a table.

## 24.22.2 Constructor & Destructor Documentation

### 24.22.2.1 CCfits::Column::InvalidRowNumber::InvalidRowNumber (const String &diag, bool silent = true)

Exception ctor, prefixes the string "FitsError: Invalid Row Number - Column: " before the specific message.

#### Parameters:

*diag* A specific diagnostic message, usually the column name.

*silent* if true, print message whether [FITS::verboseMode](#) is set or not.

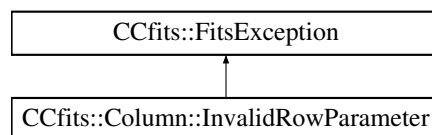
The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

## 24.23 CCfits::Column::InvalidRowParameter Class Reference

Exception thrown on incorrect row writing request.

#include <Column.h>Inheritance diagram for CCfits::Column::InvalidRowParameter::



### Public Member Functions

- [InvalidRowParameter](#) (const String &diag, bool silent=true)

*Exception ctor, prefixes the string "FitsError: row offset or length incompatible with column declaration " before the specific message.*

### 24.23.1 Detailed Description

Exception thrown on incorrect row writing request. This exception is thrown if the user requests writing more data than a fixed width row can accommodate. An exception is

thrown rather than a truncation because it is likely that the user will not otherwise realize that data loss is happening.

### 24.23.2 Constructor & Destructor Documentation

#### 24.23.2.1 CCfits::Column::InvalidRowParameter::InvalidRowParameter (const String & *diag*, bool *silent* = true)

Exception ctor, prefixes the string "FitsError: row offset or length incompatible with column declaration " before the specific message.

#### Parameters:

- diag* A specific diagnostic message, usually the column name
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

## 24.24 CCfits::Keyword Class Reference

Abstract base class defining the interface for [Keyword](#) objects.

#include <Keyword.h>

Inherited by CCfits::KeyData< T >.

### Public Member Functions

- virtual [~Keyword](#) ()  
*virtual destructor*
- virtual [Keyword](#) \* [clone](#) () const =0  
*virtual copy constructor*
- const String & [comment](#) () const  
*return the comment field of the keyword*
- fitsfile \* [fitsPointer](#) () const  
*return a pointer to the [FITS](#) file containing the parent [HDU](#).*



- const String & **name** () const  
*return the name of a keyword*
- bool **operator!=** (const **Keyword** &right) const  
*inequality operator*
- **Keyword** & **operator=** (const **Keyword** &right)  
*assignment operator*
- bool **operator==** (const **Keyword** &right) const  
*equality operator*
- template<typename T >  
void **setValue** (const T &newValue)  
*modify the value of an existing **Keyword** and write it to the file*
- template<typename T >  
T & **value** (T &val) const  
*get the keyword value*
- virtual void **write** ()  
*left in for historical reasons, this seldom needs to be called by users*

### Protected Member Functions

- **Keyword** (const String &keyname, ValueType keytype, **HDU** \*p, const String &comment="")  
***Keyword** constructor.*
- **Keyword** (const **Keyword** &right)  
*copy constructor*
- void **keytype** (ValueType value)  
*set keyword type.*
- ValueType **keytype** () const  
*return the type of a keyword*
- const **HDU** \* **parent** () const  
*return a pointer to parent **HDU**.*

### 24.24.1 Detailed Description

Abstract base class defining the interface for [Keyword](#) objects. [Keyword](#) object creation is normally performed inside [FITS](#) constructors or [FITS::read](#), [HDU::readKey](#), and [HDU::addKey](#) functions. Output is performed in [HDU::addKey](#) functions and [Keyword::setValue](#).

Keywords consists of a name, a value and a comment field. Concrete templated subclasses, `KeyData<T>`, have a data member that holds the value of keyword.

Typically, the mandatory keywords for a given [HDU](#) type are not stored as object of type [Keyword](#), but as intrinsic data types. The [Keyword](#) hierarchy is used to store user-supplied information.

### 24.24.2 Constructor & Destructor Documentation

#### 24.24.2.1 CCfits::Keyword::Keyword (const String & *keyname*, ValueType *keytype*, HDU \**p*, const String & *comment* = "") [protected]

[Keyword](#) constructor. This is the common behavior of Keywords of any type. Constructor is protected as the class is abstract.

### 24.24.3 Member Function Documentation

#### 24.24.3.1 template<typename T > void CCfits::Keyword::setValue (const T & *newValue*) [inline]

modify the value of an existing [Keyword](#) and write it to the file Parameters:

#### Parameters:

*newValue* (T) New value for the [Keyword](#)

**Allowed T types:** This must copy *newValue* to a data member of type U in the [Keyword](#) subclass `KeyData<U>` (see description for [Keyword::value](#) (T& val) for more details). To avoid compilation errors, it is generally best to provide a *newValue* of type T = type U, though the following type conversions will also be handled:

T (from <i>newValue</i> )	U (to <a href="#">Keyword</a> obj)
float	double, float
double	double, float (will lose precision)
int	double, float, int, integer string

### 24.24.3.2 template<typename T> T & CCfits::Keyword::value (T & *val*) const [inline]

get the keyword value Parameters:

#### Parameters:

*val* (T) Will be filled with the keyword value, and is also the function return value.

**Allowed T types:** CCfits stores keyword values of type U in a templated subclass of [Keyword](#), [KeyData](#)<U>. Normally U is set when reading the [Keyword](#) in from the file, and is limited to types int, double, string, bool, and complex<float>. (The exception is when the user has created and added a new [Keyword](#) using an [HDU::addKey](#) function, in which case they might have specified other types for U.) To avoid compilation errors, the user should generally try to provide a *val* of type T = type U, though there is some flexibility here as the following conversions are handled:

T (to val)	U (from <a href="#">Keyword</a> obj)
float	double (will lose precision), float, int, integer string
double	double, float, int, integer string
int	int, integer string

More conversions may be added in the future as the need arises.

### 24.24.3.3 void CCfits::Keyword::write () [virtual]

left in for historical reasons, this seldom needs to be called by users This writes the [Keyword](#) to the file, and is called internally during [HDU::addKey](#) operations or the [Keyword::setValue](#) function. It shouldn't normally need to be called explicitly.

The documentation for this class was generated from the following files:

- [Keyword.h](#)
- [Keyword.cxx](#)
- [KeywordT.h](#)

## 24.25 CCfits::FITSUtil::MatchName< T > Class Template Reference

predicate for classes that have a name attribute; match input string with instance name.

```
#include <FITSUtil.h>
```

### 24.25.1 Detailed Description

**template<class T> class CCfits::FITSUtil::MatchName< T >**

predicate for classes that have a name attribute; match input string with instance name.

Usage: MatchName<NamedClass> Ex;

list<NamedClass> ListObject;

... ..

find\_if(ListObject.begin(),ListObject().end(),bind2nd(Ex,"needle"));

Since most of the classes within CCfits are not implemented with lists, these functions are now of little direct use.

The documentation for this class was generated from the following file:

- FITSUtil.h

## 24.26 CCfits::FITSUtil::MatchNum< T > Class Template Reference

predicate for classes that have an index attribute; match input index with instance value.

#include <FITSUtil.h>

### 24.26.1 Detailed Description

**template<class T> class CCfits::FITSUtil::MatchNum< T >**

predicate for classes that have an index attribute; match input index with instance value.

Usage: MatchName<IndexedClass> Ex;

list<NamedClass> ListObject;

... ..

find\_if(ListObject.begin(),ListObject().end(),bind2nd(Ex,5));

Since most of the classes within CCfits are implemented with std::maps rather than lists, these functions are now of little direct use.

The documentation for this class was generated from the following file:

- FITSUtil.h

## 24.27 CCfits::FITSUtil::MatchPtrName< T > Class Template Reference

as for [MatchName](#), only with the input class a pointer.

```
#include <FITSUtil.h>
```

### 24.27.1 Detailed Description

```
template<class T> class CCfits::FITSUtil::MatchPtrName< T >
```

as for [MatchName](#), only with the input class a pointer.

The documentation for this class was generated from the following file:

- FITSUtil.h

## 24.28 CCfits::FITSUtil::MatchPtrNum< T > Class Template Reference

as for [MatchNum](#), only with the input class a pointer.

```
#include <FITSUtil.h>
```

### 24.28.1 Detailed Description

```
template<class T> class CCfits::FITSUtil::MatchPtrNum< T >
```

as for [MatchNum](#), only with the input class a pointer.

The documentation for this class was generated from the following file:

- FITSUtil.h

## 24.29 CCfits::FITSUtil::MatchType< T > Class Template Reference

function object that returns the [FITS](#) ValueType corresponding to an input intrinsic type

```
#include <FITSUtil.h>
```

### 24.29.1 Detailed Description

`template<typename T> class CCfits::FITSUtil::MatchType< T >`

function object that returns the [FITS](#) ValueType corresponding to an input intrinsic type. This is particularly useful inside templated class instances where calls to `cfitsio` need to supply a value type. With this function one can extract the value type from the class type.

*usage:*

```
MatchType<T> type;
```

```
ValueType dataType = type();
```

Uses run-time type information (RTTI) methods.

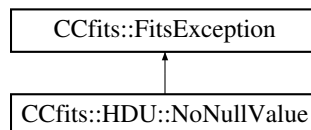
The documentation for this class was generated from the following file:

- FITSUtil.h

## 24.30 CCfits::HDU::NoNullValue Class Reference

exception to be thrown on seek errors for keywords.

`#include <HDU.h>` Inheritance diagram for `CCfits::HDU::NoNullValue`:



### Public Member Functions

- [NoNullValue](#) (const string &diag, bool silent=true)

*Exception ctor, prefixes the string "Fits Error: No Null Pixel Value specified for Image" before the specific message.*

### 24.30.1 Detailed Description

exception to be thrown on seek errors for keywords.

### 24.30.2 Constructor & Destructor Documentation

#### 24.30.2.1 CCfits::HDU::NoNullValue::NoNullValue (const string &diag, bool silent = true)

Exception ctor, prefixes the string "Fits Error: No Null Pixel Value specified for Image " before the specific message.

##### Parameters:

- diag* A specific diagnostic message, the name of the [HDU](#) if not the primary.
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

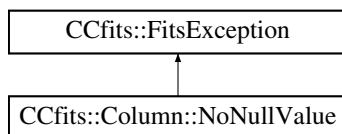
The documentation for this class was generated from the following files:

- HDU.h
- HDU.cxx

## 24.31 CCfits::Column::NoNullValue Class Reference

Exception thrown if a null value is specified without support from existing column header.

#include <Column.h>Inheritance diagram for CCfits::Column::NoNullValue::



### Public Member Functions

- [NoNullValue](#) (const String &diag, bool silent=true)  
*Exception ctor, prefixes the string "Fits Error: No null value specified for column: " before the specific message.*

#### 24.31.1 Detailed Description

Exception thrown if a null value is specified without support from existing column header. This exception is analogous to the fact that cfitsio returns a non-zero status

code if TNULLn doesn't exist an a null value (convert all input data with the null value to the TNULLn keyword) is specified. It is only relevant for integer type data (see cfitsio manual for details).

### 24.31.2 Constructor & Destructor Documentation

#### 24.31.2.1 CCfits::Column::NoNullValue::NoNullValue (const String & *diag*, bool *silent* = true)

Exception ctor, prefixes the string "Fits Error: No null value specified for column: " before the specific message.

##### Parameters:

*diag* A specific diagnostic message

*silent* if true, print message whether [FITS::verboseMode](#) is set or not.

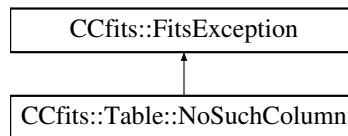
The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

## 24.32 CCfits::Table::NoSuchColumn Class Reference

Exception to be thrown on a failure to retrieve a column specified either by name or index number.

#include <Table.h> Inheritance diagram for CCfits::Table::NoSuchColumn::



### Public Member Functions

- [NoSuchColumn](#) (int index, bool silent=true)

*Exception ctor for exception thrown if the requested column (specified by name) is not present.*

- [NoSuchColumn](#) (const String &name, bool silent=true)



*Exception ctor for exception thrown if the requested column (specified by name) is not present.*

### 24.32.1 Detailed Description

Exception to be thrown on a failure to retrieve a column specified either by name or index number. When a [Table](#) object is created, the header is read and a column object created for each column defined. Thus if this exception is thrown the column requested does not exist in the [HDU](#) (note that the column can easily exist and not contain any data since the user controls whether the column will be read when the [FITS](#) object is instantiated).

It is expected that the index number calls will be primarily internal. The underlying implementation makes lookup by name more efficient.

The exception has two variants, which take either an integer or a string as parameter. These are used according to the accessor that threw them, either by name or index.

### 24.32.2 Constructor & Destructor Documentation

#### 24.32.2.1 CCfits::Table::NoSuchColumn::NoSuchColumn (const String & name, bool silent = true)

Exception ctor for exception thrown if the requested column (specified by name) is not present. Message: Fits Error: cannot find [Column](#) named: *name* is printed.

##### Parameters:

*name* the requested column name

*silent* if true, print message whether [FITS::verboseMode](#) is set or not.

#### 24.32.2.2 CCfits::Table::NoSuchColumn::NoSuchColumn (int index, bool silent = true)

Exception ctor for exception thrown if the requested column (specified by name) is not present. Message: Fits Error: column not present - [Column](#) number *index* is printed.

##### Parameters:

*index* the requested column number

*silent* if true, print message whether [FITS::verboseMode](#) is set or not.

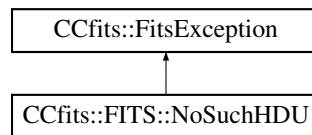
The documentation for this class was generated from the following files:

- Table.h
- Table.cxx

## 24.33 CCfits::FITS::NoSuchHDU Class Reference

exception thrown by [HDU](#) retrieval methods.

#include <FITS.h> Inheritance diagram for CCfits::FITS::NoSuchHDU::



### Public Member Functions

- [NoSuchHDU](#) (const String &diag, bool silent=true)  
*Exception ctor, prefixes the string "FITS Error: Cannot read HDU in FITS file:" before the specific message.*

#### 24.33.1 Detailed Description

exception thrown by [HDU](#) retrieval methods.

#### 24.33.2 Constructor & Destructor Documentation

##### 24.33.2.1 CCfits::FITS::NoSuchHDU::NoSuchHDU (const String &diag, bool *silent* = true)

Exception ctor, prefixes the string "FITS Error: Cannot read HDU in FITS file:" before the specific message.

#### Parameters:

*diag* A specific diagnostic message, usually the name of the extension whose read was attempted.

*silent* if true, print message whether [FITS::verboseMode](#) is set or not.

Exception to be thrown by failed seek operations

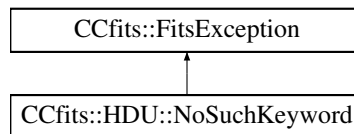
The documentation for this class was generated from the following files:

- FITS.h
- FITS.cxx

## 24.34 CCfits::HDU::NoSuchKeyword Class Reference

exception to be thrown on seek errors for keywords.

#include <Hdu.h> Inheritance diagram for CCfits::HDU::NoSuchKeyword::



### Public Member Functions

- [NoSuchKeyword](#) (const string &diag, bool silent=true)  
*Exception ctor, prefixes the string "Fits Error: Keyword not found: " before the specific message.*

#### 24.34.1 Detailed Description

exception to be thrown on seek errors for keywords.

#### 24.34.2 Constructor & Destructor Documentation

##### 24.34.2.1 CCfits::HDU::NoSuchKeyword::NoSuchKeyword (const string &diag, bool silent = true)

Exception ctor, prefixes the string "Fits Error: Keyword not found: " before the specific message.

#### Parameters:

*diag* A specific diagnostic message, usually the name of the keyword requested.

*silent* if true, print message whether [FITS::verboseMode](#) is set or not.

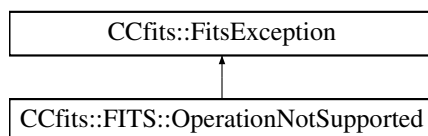
The documentation for this class was generated from the following files:

- HDU.h
- HDU.cxx

## 24.35 CCfits::FITS::OperationNotSupported Class Reference

thrown for unsupported operations, such as attempted to select rows from an image extension.

#include <FITS.h> Inheritance diagram for CCfits::FITS::OperationNotSupported::



### Public Member Functions

- [OperationNotSupported](#) (const String &msg, bool silent=true)  
*Exception ctor, prefixes the string "FITS Error: Operation not supported:" before the specific message.*

#### 24.35.1 Detailed Description

thrown for unsupported operations, such as attempted to select rows from an image extension.

#### 24.35.2 Constructor & Destructor Documentation

##### 24.35.2.1 CCfits::FITS::OperationNotSupported::OperationNotSupported (const String & msg, bool silent = true)

Exception ctor, prefixes the string "FITS Error: Operation not supported:" before the specific message.

**Parameters:**

- msg* A specific diagnostic message.
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

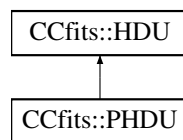
The documentation for this class was generated from the following files:

- FITS.h
- FITS.cxx

**24.36 CCfits::PHDU Class Reference**

class representing the primary [HDU](#) for a [FITS](#) file.

`#include <PHDU.h>` Inheritance diagram for CCfits::PHDU::

**Public Member Functions**

- virtual [~PHDU](#) ()  
*destructor*
- virtual [PHDU](#) \* [clone](#) (FITSBase \*p) const =0  
*virtual copy constructor, to be implemented in subclasses.*
- template<typename S >  
void [read](#) (std::valarray< S > &image, const std::vector< long > &firstVertex,  
const std::vector< long > &lastVertex, const std::vector< long > &stride, S  
\*nullValue)  
*read an image subset into valarray image, processing null values*
- template<typename S >  
void [read](#) (std::valarray< S > &image, const std::vector< long > &firstVertex,  
const std::vector< long > &lastVertex, const std::vector< long > &stride)  
*read an image subset*
- template<typename S >  
void [read](#) (std::valarray< S > &image, const std::vector< long > &first, long  
nElements, S \*nullValue)

*read part of an image array, processing null values.*

- template<typename S >  
void [read](#) (std::valarray< S > &image, const std::vector< long > &first, long nElements)

*read an image section starting at a location specified by an n-tuple*

- template<typename S >  
void [read](#) (std::valarray< S > &image, long first, long nElements, S \*nullValue)

*read part of an image array, processing null values.*

- template<typename S >  
void [read](#) (std::valarray< S > &image, long first, long nElements)

*read an image section starting at a specified pixel*

- virtual void [readData](#) (bool readFlag=false, const std::vector< String > &keys=std::vector< String >())=0

*read primary [HDU](#) data*

- virtual double [scale](#) () const

*return the BSCALE keyword value*

- virtual void [scale](#) (double value)

*set the BSCALE keyword value for images (see warning for images of int type)*

- template<typename S >  
void [write](#) (const std::vector< long > &firstVertex, const std::vector< long > &lastVertex, const std::vector< long > &stride, const std::valarray< S > &data)

*write a subset (generalize slice) of data to the image*

- template<typename S >  
void [write](#) (long first, long nElements, const std::valarray< S > &data)

*write array starting from specified pixel number, without undefined data processing*

- template<typename S >  
void [write](#) (const std::vector< long > &first, long nElements, const std::valarray< S > &data)

*write array starting from specified n-tuple, without undefined data processing*

- template<typename S >  
void [write](#) (long first, long nElements, const std::valarray< S > &data, S \*nullValue)

*write array to image starting with a specified pixel and allowing undefined data to be processed*

- template<typename S >  
void [write](#) (const std::vector< long > &first, long nElements, const std::valarray< S > &data, S \*nullValue)

*Write a set of pixels to an image extension with the first pixel specified by an n-tuple, processing undefined data.*

- virtual double [zero](#) () const  
*return the BZERO keyword value*
- virtual void [zero](#) (double value)  
*set the BZERO keyword value for images (see warning for images of int type)*

### Protected Member Functions

- [PHDU](#) (FITSBase \*p=0)  
*Reading Primary HDU constructor.*
- [PHDU](#) (FITSBase \*p, int bpix, int naxis, const std::vector< long > &axes)  
*Writing Primary HDU constructor, called by PrimaryHDU<T> class.*
- [PHDU](#) (const [PHDU](#) &right)  
*copy constructor*
- virtual void [initRead](#) ()

#### 24.36.1 Detailed Description

class representing the primary [HDU](#) for a [FITS](#) file. A [PHDU](#) object is automatically instantiated and added to a [FITS](#) object when a [FITS](#) file is accessed in any way. If a new file is created without specifying the data type for the header, CCfits assumes that the file is to be used for table extensions and creates a dummy header. [PHDU](#) instances are *only* created by [FITS](#) ctors. In the first release of CCfits, the Primary cannot be changed once declared.

[PHDU](#) and [ExtHDU](#) provide the same interface to writing images: multiple overloads of the templated [PHDU::read](#) and [PHDU::write](#) operations provide for (a) writing image data specified in a number of ways [C-array, std::vector, std::valarray] and with input location specified by initial pixel, by n-tuple, and by rectangular subset [generalized slice]; (b) reading image data specified similarly to the write options into a std::valarray.

### Todo

Implement functions that allow replacement of the primary image

## 24.36.2 Constructor & Destructor Documentation

### 24.36.2.1 CCfits::PHDU::~~PHDU () [virtual]

destructor

Destructor

### 24.36.2.2 CCfits::PHDU::PHDU (const PHDU & *right*) [protected]

copy constructor required for cloning primary HDUs when copying [FITS](#) files.

### 24.36.2.3 CCfits::PHDU::PHDU (FITSBase \* *p*, int *bpix*, int *naxis*, const std::vector< long > & *axes*) [protected]

Writing Primary [HDU](#) constructor, called by PrimaryHDU<T> class. Constructor used for creating new [PHDU](#) (i.e. for writing data to [FITS](#)). also doubles as default constructor since all arguments have default values, which are passed to the [HDU](#) constructor

### 24.36.2.4 CCfits::PHDU::PHDU (FITSBase \* *p* = 0) [protected]

Reading Primary [HDU](#) constructor. Constructor used when reading the primary [HDU](#) from an existing file. Does nothing except initialize, with the real work done by the subclass PrimaryHDU<T>.

## 24.36.3 Member Function Documentation

### 24.36.3.1 void CCfits::PHDU::initRead () [protected, virtual]

Read image header and update fits pointer accordingly.



Private: called by ctor.

Implements [CCfits::HDU](#).

**24.36.3.2** `template<typename S> void CCfits::PHDU::read (std::valarray< S> & image, const std::vector< long> & firstVertex, const std::vector< long> & lastVertex, const std::vector< long> & stride, S * nullValue) [inline]`

read an image subset into valarray *image*, processing null values. The image subset is defined by two vertices and a stride indicating the 'denseness' of the values to be picked in each dimension (a stride = (1,1,1,...) means picking every pixel in every dimension, whereas stride = (2,2,2,...) means picking every other value in each dimension.

**24.36.3.3** `template<typename S> void CCfits::PHDU::read (std::valarray< S> & image, const std::vector< long> & first, long nElements, S * nullValue) [inline]`

read part of an image array, processing null values. As above except for

**Parameters:**

*first* a vector<long> representing an n-tuple giving the coordinates in the image of the first pixel.

**24.36.3.4** `template<typename S> void CCfits::PHDU::read (std::valarray< S> & image, long first, long nElements, S * nullValue) [inline]`

read part of an image array, processing null values. Implicit data conversion is supported (i.e. user does not need to know the type of the data stored. A WrongExtension-Type extension is thrown if \*this is not an image.

**Parameters:**

*image* The receiving container, a std::valarray reference

*first* The first pixel from the array to read [a long value]

*nElements* The number of values to read

*nullValue* A pointer containing the value in the table to be considered as undefined. See cfitsio for details

**24.36.3.5** `void CCfits::PHDU::readData (bool readFlag = false, const std::vector< String > & keys = std::vector<String> ())`  
**[pure virtual]**

read primary [HDU](#) data Called by [FITS](#) ctor, not intended for general use. parameters control how much gets read on initialization. An abstract function, implemented in the subclasses.

**Parameters:**

*readFlag* read the image data if true

*key* a vector of strings of keyword names to be read from the primary [HDU](#)

**24.36.3.6** `void CCfits::PHDU::scale (double value)` **[virtual]**

set the BSCALE keyword value for images (see warning for images of int type) For primary HDUs and image extensions, this will add (or update) the BSCALE keyword in the header. The new setting will affect future image array read/writes as described in section 4.7 Data Scaling of the CFITSIO manual. For table extensions this function does nothing.

WARNING: If the image contains **integer-type data** (as indicated by the [bitpix\(\)](#) return value), the new scale and zero value combination must not be such that the scaled data would require a floating-point type (this uses the CFITSIO function `fits_get_img_equivtype` to make the determination). If this situation occurs, the function will throw a [FitsException](#).

Reimplemented from [CCfits::HDU](#).

**24.36.3.7** `template<typename S > void CCfits::PHDU::write (const std::vector< long > & firstVertex, const std::vector< long > & lastVertex, const std::vector< long > & stride, const std::valarray< S > & data)` **[inline]**

write a subset (generalize slice) of data to the image A generalized slice/subset is a subset of the image (e.g. one plane of a data cube of size <= the dimension of the cube). It is specified by two opposite vertices. The equivalent cfitsio call does not support undefined data processing so there is no version that allows a null value to be specified.

**Parameters:**

*firstVertex* The coordinates specifying lower and upper vertices of the n-dimensional slice

*lastVertex*

*stride* Pixels to skip in each to dimension, e.g. stride = (1,1,1,...) means picking every pixel in every dimension, whereas stride = (2,2,2,...) means picking every other value in each dimension.

*data* The data to be written

**24.36.3.8** `template<typename S > void CCfits::PHDU::write (long first,  
long nElements, const std::valarray< S > & data, S * nullValue)  
[inline]`

write array to image starting with a specified pixel and allowing undefined data to be processed parameters after the first are as for version with n-tuple specifying first element. these two version are equivalent, except that it is possible for the first pixel number to exceed the range of 32-bit integers, which is how long datatype is commonly implemented.

**24.36.3.9** `template<typename S > void CCfits::PHDU::write (const  
std::vector< long > & first, long nElements, const std::valarray< S  
> & data, S * nullValue) [inline]`

Write a set of pixels to an image extension with the first pixel specified by an n-tuple, processing undefined data. All the overloaded versions of [PHDU::write](#) perform operations on \*this if it is an image and throw a WrongExtensionType exception if not. Where appropriate, alternate versions allow undefined data to be processed

**Parameters:**

*first* an n-tuple of dimension equal to the image dimension specifying the first pixel in the range to be written

*nElements* number of pixels to be written

*data* array of data to be written

*nullValue* pointer to null value (data with this value written as undefined; needs the BLANK keyword to have been specified).

### 24.36.3.10 void CCfits::PHDU::zero (double value) [virtual]

set the BZERO keyword value for images (see warning for images of int type) For primary HDUs and image extensions, this will add (or update) the BZERO keyword in the header. The new setting will affect future image array read/writes as described in section 4.7 Data Scaling of the CFITSIO manual. For table extensions this function does nothing.

WARNING: If the image contains **integer-type data** (as indicated by the [bitpix\(\)](#) return value), the new scale and zero value combination must not be such that the scaled data would require a floating-point type (this uses the CFITSIO function `fits_get_img_equivtype` to make the determination). If this situation occurs, the function will throw a [FitsException](#).

Reimplemented from [CCfits::HDU](#).

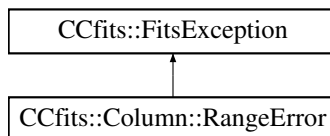
The documentation for this class was generated from the following files:

- PHDU.h
- PHDU.cxx
- PHDUT.h

## 24.37 CCfits::Column::RangeError Class Reference

exception to be thrown for inputs that cause range errors in column read operations.

#include <Column.h> Inheritance diagram for CCfits::Column::RangeError::



### Public Member Functions

- [RangeError](#) (const String &msg, bool silent=true)  
*Exception ctor, prefixes the string "FitsError: Range error in operation " before the specific message.*

### 24.37.1 Detailed Description

exception to be thrown for inputs that cause range errors in column read operations. Range errors here mean (last < first) in a request to read a range of rows.

### 24.37.2 Constructor & Destructor Documentation

#### 24.37.2.1 CCfits::Column::RangeError::RangeError (const String & *msg*, bool *silent* = `true`)

Exception ctor, prefixes the string "FitsError: Range error in operation " before the specific message.

#### Parameters:

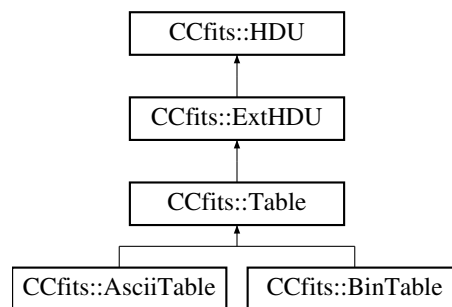
- msg* A specific diagnostic message
- silent* if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

## 24.38 CCfits::Table Class Reference

#include <Table.h> Inheritance diagram for CCfits::Table::



#### Classes

- class [NoSuchColumn](#)

*Exception to be thrown on a failure to retrieve a column specified either by name or index number.*

### Public Member Functions

- **Table** (const **Table** &right)  
*copy constructor*
- virtual **~Table** ()  
*destructor*
- virtual std::map< string, **Column** \* > & **column** ()  
*return a reference to the array containing the columns.*
- virtual const std::map< string, **Column** \* > & **column** () const  
*return a reference to the array containing the columns.*
- virtual **Column** & **column** (int colIndex) const  
*return a reference to the column identified by colIndex*
- virtual **Column** & **column** (const String &colName, bool caseSensitive=true) const  
*return a reference to a **Table** column specified by name.*
- virtual void **deleteColumn** (const String &columnName)  
*delete a column in a **Table** extension by name.*
- void **deleteRows** (const std::vector< long > &rowList)  
*delete a set of rows in the table specified by an input array.*
- void **deleteRows** (long first, long number=1)  
*delete a range of rows in a table.*
- virtual long **getRowsize** () const  
*return the optimal number of rows to read or write at a time*
- void **insertRows** (long first, long number=1)  
*insert empty rows into the table*
- virtual int **numCols** () const  
*return the number of Columns in the **Table** (the **TFIELDS** keyword).*

- void [rows](#) (long numRows)  
*set the number of rows in the [Table](#).*
- virtual long [rows](#) () const  
*return the number of rows in the table (NAXIS2).*
- void [updateRows](#) ()  
*update the number of rows in the table*

### Protected Member Functions

- [Table](#) (FITSBase \*p, HduType xtype, int number)  
*[Table](#) constructor for getting [Tables](#) by number.*
- [Table](#) (FITSBase \*p, HduType xtype, const String &hduName=String(""), int version=1)  
*Constructor to be called by operations that read [Table](#) specified by hduName and version.*
- [Table](#) (FITSBase \*p, HduType xtype, const String &hduName, int rows, const std::vector< String > &columnName, const std::vector< String > &columnFmt, const std::vector< String > &columnUnit=std::vector< String >(), int version=1)  
*Constructor to be used for creating new HDUs.*
- void [init](#) (bool readFlag=false, const std::vector< String > &keys=std::vector< String >())
- void [numCols](#) (int value)  
*set the number of Columns in the [Table](#)*
- virtual void [setColumn](#) (const String &colname, [Column](#) \*value)  
*set the column with name colname to the input value.*

#### 24.38.1 Detailed Description

[Table](#) is the abstract common interface to Binary and Ascii [Table](#) HDUs.

[Table](#) is a subclass of [ExtHDU](#) that contains an associative array of [Column](#) objects. It implements methods for reading and writing columns

## 24.38.2 Constructor & Destructor Documentation

**24.38.2.1** `CCfits::Table::Table (FITSBase * p, HduType xtype, const String & hduName, int rows, const std::vector< String > & columnName, const std::vector< String > & columnFmt, const std::vector< String > & columnUnit = std::vector<String> (), int version = 1)`  
**[protected]**

Constructor to be used for creating new HDUs.

### Parameters:

- p* The [FITS](#) file in which to place the new [HDU](#)
- xtype* An HduType enumerator defined in CCfits.h for type of table (AsciiTbl or BinaryTbl)
- hduName* The name of this [HDU](#) extension
- rows* The number of rows in the new [HDU](#) (the value of the NAXIS2 keyword).
- columnName* a vector of names for the columns.
- columnFmt* the format strings for the columns
- columnUnit* the units for the columns.
- version* a version number

**24.38.2.2** `CCfits::Table::Table (FITSBase * p, HduType xtype, int number)`  
**[protected]**

[Table](#) constructor for getting Tables by number. Necessary since EXTNAME is a reserved not required keyword, and users may thus read [FITS](#) files without an extension name. Since an [HDU](#) is completely specified by extension number, this is part of the public interface.

## 24.38.3 Member Function Documentation

**24.38.3.1** `std::map< string, Column * > & CCfits::Table::column ()`  
**[inline, virtual]**

return a reference to the array containing the columns. To be used in the implementation of subclasses.



**24.38.3.2** `const std::map< string, Column * > & CCfits::Table::column ()  
const [inline, virtual]`

return a reference to the array containing the columns. This public version might be used to query the size of the column container in a routine that manipulates column table data.

Reimplemented from [CCfits::ExtHDU](#).

**24.38.3.3** `Column & CCfits::Table::column (int colIndex) const [virtual]`

return a reference to the column identified by colIndex Throws [NoSuchColumn](#) if the index is out of range -index must satisfy (1 <= index <= [numCols\(\)](#) ).

N.B. the column number is assigned as 1-based, as in FORTRAN rather than 0-based as in C.

**Exceptions:**

[Table::NoSuchColumn](#) passes colIndex to the diagnostic message printed when the exception is thrown

Reimplemented from [CCfits::ExtHDU](#).

**24.38.3.4** `Column & CCfits::Table::column (const String & colName, bool  
caseSensitive = true) const [virtual]`

return a reference to a [Table](#) column specified by name. If the *caseSensitive* parameter is set to false, the search will be case-insensitive. The overridden base class implementation [ExtHDU::column](#) throws an exception, which is thus the action to be taken if self is an image extension

**Exceptions:**

*WrongExtensionType* see above

Reimplemented from [CCfits::ExtHDU](#).

**24.38.3.5** `void CCfits::Table::deleteColumn (const String & columnName)  
[virtual]`

delete a column in a [Table](#) extension by name.

**Parameters:**

*columnName* The name of the column to be deleted.

**Exceptions:**

*WrongExtensionType* if extension is an image.

Reimplemented from [CCfits::ExtHDU](#).

### 24.38.3.6 void CCfits::Table::deleteRows (const std::vector< long > & rowlist)

delete a set of rows in the table specified by an input array.

**Parameters:**

*rowlist* The vector of row numbers to be deleted.

**Exceptions:**

*FitsError* thrown if the underlying cfitsio call fails to return without error.

### 24.38.3.7 void CCfits::Table::deleteRows (long first, long number = 1)

delete a range of rows in a table. In both this and the overloaded version which allows a selection of rows to be deleted, the cfitsio library is called first to perform the operation on the disk file, and then the [FITS](#) object is updated.

**Parameters:**

*first* the start row of the range

*number* the number of rows to delete; defaults to 1.

**Exceptions:**

*FitsError* thrown if the cfitsio call fails to return without error.

### 24.38.3.8 long CCfits::Table::getRowSize () const [virtual]

return the optimal number of rows to read or write at a time A wrapper for the CFIT-SIO function `fits_get_rowsize`, useful for obtaining maximum I/O efficiency. This will throw if it is not called for a [Table](#) extension.

Reimplemented from [CCfits::ExtHDU](#).

### 24.38.3.9 void CCfits::Table::init (bool *readFlag* = false, const std::vector<String> & *keys* = std::vector<String>()) [protected]

"Late Constructor." wrap-up of calls needed to construct a table. Reads header information and sets up the array of column objects in the table.

Protected function, provided to allow the implementation of extensions of the library.

### 24.38.3.10 void CCfits::Table::insertRows (long *first*, long *number* = 1)

insert empty rows into the table

#### Parameters:

- first* the start row of the range
- number* the number of rows to insert.

#### Exceptions:

[FitsError](#) thrown if the underlying cfitsio call fails to return without error.

### 24.38.3.11 void CCfits::Table::updateRows ()

update the number of rows in the table Called to force the [Table](#) to reset its internal "rows" attribute. public, but is called when needed internally.

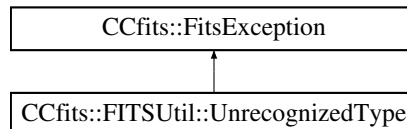
The documentation for this class was generated from the following files:

- Table.h
- Table.cxx

## 24.39 CCfits::FITSUtil::UnrecognizedType Class Reference

exception thrown by [MatchType](#) if it encounters data type incompatible with cfitsio.

#include <FITSUtil.h> Inheritance diagram for CCfits::FITSUtil::UnrecognizedType::



### 24.39.1 Detailed Description

exception thrown by [MatchType](#) if it encounters data type incompatible with cfitsio.

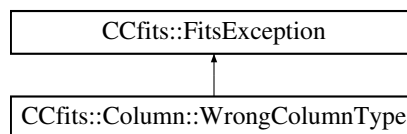
The documentation for this class was generated from the following files:

- FITSUtil.h
- FITSUtil.cxx

## 24.40 CCfits::Column::WrongColumnType Class Reference

Exception thrown on attempting to access a scalar column as vector data.

#include <Column.h> Inheritance diagram for CCfits::Column::WrongColumnType::



### Public Member Functions

- [WrongColumnType](#) (const String &diag, bool silent=true)

*Exception ctor, prefixes the string "FitsError: Attempt to return scalar data from vector column, or vice versa - Column: " before the specific message.*

### 24.40.1 Detailed Description

Exception thrown on attempting to access a scalar column as vector data. This exception will be thrown if the user tries to call a read/write operation with a signature appropriate for a vector column on a scalar column, or vice versa. For example in the case of write operations, the vector versions require the specification of (a) a number of rows to write over, (b) a vector of lengths to write to each row or (c) a subset specification. The scalar versions only require a number of rows if the input array is a plain C-array, otherwise the range to be written is the size of the input vector.

### 24.40.2 Constructor & Destructor Documentation

#### 24.40.2.1 CCfits::Column::WrongColumnType::WrongColumnType (const String & *diag*, bool *silent* = true)

Exception ctor, prefixes the string "FitsError: Attempt to return scalar data from vector column, or vice versa - Column: " before the specific message.

#### Parameters:

*diag* A specific diagnostic message, usually the column name.

*silent* if true, print message whether [FITS::verboseMode](#) is set or not.

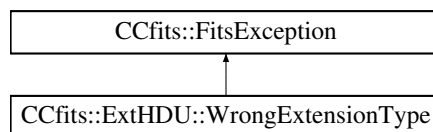
The documentation for this class was generated from the following files:

- Column.h
- Column.cxx

## 24.41 CCfits::ExtHDU::WrongExtensionType Class Reference

Exception to be thrown on unmatched extension types.

```
#include <ExtHDU.h>
Inheritance diagram for CCfits::ExtHDU::WrongExtensionType::
```



## Public Member Functions

- [WrongExtensionType](#) (const String &msg, bool silent=true)

*Exception ctor, prefixes the string "Fits Error: wrong extension type" before the specific message.*

### 24.41.1 Detailed Description

Exception to be thrown on unmatched extension types. This exception is to be thrown if the user requested a particular extension and it does not correspond to the expected type.

### 24.41.2 Constructor & Destructor Documentation

#### 24.41.2.1 CCfits::ExtHDU::WrongExtensionType::WrongExtensionType (const String & msg, bool silent = true)

Exception ctor, prefixes the string "Fits Error: wrong extension type" before the specific message.

#### Parameters:

*msg* A specific diagnostic message

*silent* if true, print message whether [FITS::verboseMode](#) is set or not.

The documentation for this class was generated from the following files:

- ExtHDU.h
- ExtHDU.cxx

## Index

- ~PHDU
  - CCfits::PHDU, [134](#)
- addColumn
  - CCfits::AsciiTable, [41](#)
  - CCfits::BinTable, [46](#)
  - CCfits::ExtHDU, [73](#)
- addImage
  - CCfits::FITS, [88](#)
- addKey
  - CCfits::HDU, [103](#)
- addNullValue
  - CCfits::Column, [57](#)
- addTable
  - CCfits::FITS, [88](#)
- AsciiTable
  - CCfits::AsciiTable, [40](#), [41](#)
- axis
  - CCfits::HDU, [104](#)
- BinTable
  - CCfits::BinTable, [45](#), [46](#)
- bitpix
  - CCfits::HDU, [104](#)
- CantCreate
  - CCfits::FITS::CantCreate, [48](#)
- CantOpen
  - CCfits::FITS::CantOpen, [49](#)
- CCfits::AsciiTable, [38](#)
  - addColumn, [41](#)
  - AsciiTable, [40](#), [41](#)
  - readData, [42](#)
- CCfits::BinTable, [44](#)
  - addColumn, [46](#)
  - BinTable, [45](#), [46](#)
  - readData, [47](#)
- CCfits::Column, [50](#)
  - addNullValue, [57](#)
  - Column, [56](#)
  - dimen, [57](#)
  - display, [57](#)
  - format, [58](#)
  - getNullValue, [58](#)
  - read, [58](#), [59](#)
  - readArrays, [60](#)
  - readData, [60](#)
  - resetRead, [61](#)
  - rows, [61](#)
  - scale, [61](#)
  - write, [61–66](#)
  - writeArrays, [66](#)
  - zero, [67](#)
- CCfits::Column::InsufficientElements,
  - [111](#)
  - InsufficientElements, [111](#)
- CCfits::Column::InvalidDataType, [112](#)
  - InvalidDataType, [113](#)
- CCfits::Column::InvalidNumberOfRows,
  - [115](#)
  - InvalidNumberOfRows, [116](#)
- CCfits::Column::InvalidRowNumber,
  - [116](#)
  - InvalidRowNumber, [117](#)
- CCfits::Column::InvalidRowParameter,
  - [117](#)
  - InvalidRowParameter, [118](#)
- CCfits::Column::NoNullValue, [125](#)
  - NoNullValue, [126](#)
- CCfits::Column::RangeError, [138](#)
  - RangeError, [139](#)
- CCfits::Column::WrongColumnType,
  - [146](#)
  - WrongColumnType, [147](#)
- CCfits::ExtHDU, [68](#)
  - addColumn, [73](#)
  - column, [74](#)
  - deleteColumn, [74](#)
  - ExtHDU, [73](#)
  - getRowsize, [75](#)
  - makeThisCurrent, [75](#)
  - numCols, [75](#)
  - read, [75](#), [76](#)
  - readHduName, [77](#)
  - rows, [77](#)

- write, [77, 78](#)
- xtension, [78](#)
- CCfits::ExtHDU::WrongExtensionType, [147](#)
- WrongExtensionType, [148](#)
- CCfits::FITS, [79](#)
- addImage, [88](#)
- addTable, [88](#)
- copy, [89](#)
- currentExtensionName, [89](#)
- deleteExtension, [89, 90](#)
- destroy, [90](#)
- extension, [90](#)
- filter, [90](#)
- FITS, [84–87](#)
- flush, [91](#)
- getTileDimensions, [91](#)
- read, [91, 92](#)
- setCompressionType, [93](#)
- setNoiseBits, [93](#)
- setTileDimensions, [93](#)
- verboseMode, [94](#)
- CCfits::FITS::CantCreate, [48](#)
- CantCreate, [48](#)
- CCfits::FITS::CantOpen, [49](#)
- CantOpen, [49](#)
- CCfits::FITS::NoSuchHDU, [128](#)
- NoSuchHDU, [128](#)
- CCfits::FITS::OperationNotSupported, [130](#)
- OperationNotSupported, [130](#)
- CCfits::FitsError, [94](#)
- FitsError, [95](#)
- CCfits::FitsException, [95](#)
- FitsException, [97](#)
- message, [97](#)
- CCfits::FitsFatal, [97](#)
- FitsFatal, [98](#)
- CCfits::FITSUtil::auto\_array\_ptr, [42](#)
- CCfits::FITSUtil::CAarray, [47](#)
- CCfits::FITSUtil::CVAarray, [67](#)
- CCfits::FITSUtil::CVarray, [68](#)
- CCfits::FITSUtil::MatchName, [121](#)
- CCfits::FITSUtil::MatchNum, [122](#)
- CCfits::FITSUtil::MatchPtrName, [123](#)
- CCfits::FITSUtil::MatchPtrNum, [123](#)
- CCfits::FITSUtil::MatchType, [123](#)
- CCfits::FITSUtil::UnrecognizedType, [146](#)
- CCfits::HDU, [98](#)
- addKey, [103](#)
- axis, [104](#)
- bitpix, [104](#)
- copyAllKeys, [104](#)
- deleteKey, [104](#)
- getChecksum, [104](#)
- getComments, [105](#)
- getHistory, [105](#)
- keywordCategories, [105](#)
- makeThisCurrent, [105](#)
- readAllKeys, [105](#)
- readKey, [106](#)
- readKeys, [106](#)
- scale, [106](#)
- suppressScaling, [106](#)
- updateChecksum, [107](#)
- verifyChecksum, [107](#)
- writeChecksum, [107](#)
- writeComment, [107](#)
- writeHistory, [108](#)
- zero, [108](#)
- CCfits::HDU::InvalidExtensionType, [113](#)
- InvalidExtensionType, [114](#)
- CCfits::HDU::InvalidImageDataType, [114](#)
- InvalidImageDataType, [115](#)
- CCfits::HDU::NoNullValue, [124](#)
- NoNullValue, [125](#)
- CCfits::HDU::NoSuchKeyword, [129](#)
- NoSuchKeyword, [129](#)
- CCfits::ImageExt, [108](#)
- readData, [110](#)
- scale, [110](#)
- zero, [110](#)
- CCfits::Keyword, [118](#)
- Keyword, [120](#)
- setValue, [120](#)
- value, [120](#)
- write, [121](#)
- CCfits::PHDU, [131](#)
- ~PHDU, [134](#)
- initRead, [134](#)



- PHDU, [134](#)
- read, [135](#)
- readData, [135](#)
- scale, [136](#)
- write, [136](#), [137](#)
- zero, [137](#)
- CCfits::Table, [139](#)
  - column, [142](#), [143](#)
  - deleteColumn, [143](#)
  - deleteRows, [144](#)
  - getRowsize, [144](#)
  - init, [145](#)
  - insertRows, [145](#)
  - Table, [142](#)
  - updateRows, [145](#)
- CCfits::Table::NoSuchColumn, [126](#)
- NoSuchColumn, [127](#)
- Column
  - CCfits::Column, [56](#)
- column
  - CCfits::ExtHDU, [74](#)
  - CCfits::Table, [142](#), [143](#)
- copy
  - CCfits::FITS, [89](#)
- copyAllKeys
  - CCfits::HDU, [104](#)
- currentExtensionName
  - CCfits::FITS, [89](#)
- deleteColumn
  - CCfits::ExtHDU, [74](#)
  - CCfits::Table, [143](#)
- deleteExtension
  - CCfits::FITS, [89](#), [90](#)
- deleteKey
  - CCfits::HDU, [104](#)
- deleteRows
  - CCfits::Table, [144](#)
- destroy
  - CCfits::FITS, [90](#)
- dimen
  - CCfits::Column, [57](#)
- display
  - CCfits::Column, [57](#)
- extension
  - CCfits::FITS, [90](#)
- ExtHDU
  - CCfits::ExtHDU, [73](#)
- filter
  - CCfits::FITS, [90](#)
- FITS
  - CCfits::FITS, [84–87](#)
- FITS Exceptions, [36](#)
- FitsError
  - CCfits::FitsError, [95](#)
- FitsException
  - CCfits::FitsException, [97](#)
- FitsFatal
  - CCfits::FitsFatal, [98](#)
- FITSUtil, [38](#)
- flush
  - CCfits::FITS, [91](#)
- format
  - CCfits::Column, [58](#)
- getChecksum
  - CCfits::HDU, [104](#)
- getComments
  - CCfits::HDU, [105](#)
- getHistory
  - CCfits::HDU, [105](#)
- getNullValue
  - CCfits::Column, [58](#)
- getRowsize
  - CCfits::ExtHDU, [75](#)
  - CCfits::Table, [144](#)
- getTileDimensions
  - CCfits::FITS, [91](#)
- init
  - CCfits::Table, [145](#)
- initRead
  - CCfits::PHDU, [134](#)
- insertRows
  - CCfits::Table, [145](#)
- InsufficientElements
  - CCfits::Column::InsufficientElements, [111](#)
- InvalidDataType

- CCfits::Column::InvalidDataType, [113](#)
- InvalidExtensionType
  - CCfits::HDU::InvalidExtensionType, [114](#)
- InvalidImageDataType
  - CCfits::HDU::InvalidImageDataType, [115](#)
- InvalidNumberOfRows
  - CCfits::Column::InvalidNumberOfRows, [116](#)
- InvalidRowNumber
  - CCfits::Column::InvalidRowNumber, [117](#)
- InvalidRowParameter
  - CCfits::Column::InvalidRowParameter, [118](#)
- Keyword
  - CCfits::Keyword, [120](#)
- keywordCategories
  - CCfits::HDU, [105](#)
- makeThisCurrent
  - CCfits::ExtHDU, [75](#)
  - CCfits::HDU, [105](#)
- message
  - CCfits::FitsException, [97](#)
- NoNullValue
  - CCfits::Column::NoNullValue, [126](#)
  - CCfits::HDU::NoNullValue, [125](#)
- NoSuchColumn
  - CCfits::Table::NoSuchColumn, [127](#)
- NoSuchHDU
  - CCfits::FITS::NoSuchHDU, [128](#)
- NoSuchKeyword
  - CCfits::HDU::NoSuchKeyword, [129](#)
- numCols
  - CCfits::ExtHDU, [75](#)
- OperationNotSupported
  - CCfits::FITS::OperationNotSupported, [130](#)
- PHDU
  - CCfits::PHDU, [134](#)
- RangeError
  - CCfits::Column::RangeError, [139](#)
- read
  - CCfits::Column, [58](#), [59](#)
  - CCfits::ExtHDU, [75](#), [76](#)
  - CCfits::FITS, [91](#), [92](#)
  - CCfits::PHDU, [135](#)
- readAllKeys
  - CCfits::HDU, [105](#)
- readArrays
  - CCfits::Column, [60](#)
- readData
  - CCfits::AsciiTable, [42](#)
  - CCfits::BinTable, [47](#)
  - CCfits::Column, [60](#)
  - CCfits::ImageExt, [110](#)
  - CCfits::PHDU, [135](#)
- readHduName
  - CCfits::ExtHDU, [77](#)
- readKey
  - CCfits::HDU, [106](#)
- readKeys
  - CCfits::HDU, [106](#)
- resetRead
  - CCfits::Column, [61](#)
- rows
  - CCfits::Column, [61](#)
  - CCfits::ExtHDU, [77](#)
- scale
  - CCfits::Column, [61](#)
  - CCfits::HDU, [106](#)
  - CCfits::ImageExt, [110](#)
  - CCfits::PHDU, [136](#)
- setCompressionType
  - CCfits::FITS, [93](#)
- setNoiseBits
  - CCfits::FITS, [93](#)
- setTileDimensions
  - CCfits::FITS, [93](#)
- setValue
  - CCfits::Keyword, [120](#)
- suppressScaling
  - CCfits::HDU, [106](#)
- Table

---

- CCfits::Table, [142](#)
- updateChecksum
  - CCfits::HDU, [107](#)
- updateRows
  - CCfits::Table, [145](#)
- value
  - CCfits::Keyword, [120](#)
- verboseMode
  - CCfits::FITS, [94](#)
- verifyChecksum
  - CCfits::HDU, [107](#)
- write
  - CCfits::Column, [61–66](#)
  - CCfits::ExtHDU, [77, 78](#)
  - CCfits::Keyword, [121](#)
  - CCfits::PHDU, [136, 137](#)
- writeArrays
  - CCfits::Column, [66](#)
- writeChecksum
  - CCfits::HDU, [107](#)
- writeComment
  - CCfits::HDU, [107](#)
- writeHistory
  - CCfits::HDU, [108](#)
- WrongColumnType
  - CCfits::Column::WrongColumnType,  
[147](#)
- WrongExtensionType
  - CCfits::ExtHDU::WrongExtensionType,  
[148](#)
- xtension
  - CCfits::ExtHDU, [78](#)
- zero
  - CCfits::Column, [67](#)
  - CCfits::HDU, [108](#)
  - CCfits::ImageExt, [110](#)
  - CCfits::PHDU, [137](#)